По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-038-3, название «Программирование на C#, 2-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Programming C#

Second Edition

Jesse Liberty

O'REILLY®

Программирование на C#

Второе издание

Джесс Либерти

Символ®

Санкт-Петербург—Москва
2003

Джесс Либерти

Программирование на C#, 2-е издание

Перевод С. Иноземцева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научные редакторы	*А. Королев, Н. Секунов*
Редакторы	*А. Лосев, В. Овчинников*
Корректор	*С. Беляева*
Верстка	*Н. Гриценко, А. Дорошенко*

Либерти Д.
Программирование на C#. – Пер. с англ. – СПб: Символ-Плюс, 2003. – 688 с., ил.
ISBN 5-93286-038-3

Созданный в Microsoft специально для новой платформы .NET, язык C# прост, безопасен и нацелен на работу в Интернете. Книга «Программирование на C#» поможет вам быстро приступить к практической разработке веб- и Windows-приложений на платформе .NET, работающих как на локальном компьютере, так и в Интернете.

В первой части книги обсуждаются основы языка и такие понятия ООП, как классы и объекты, наследование и полиморфизм, перегрузка операций, структуры и интерфейсы, массивы, индексаторы и коллекции, строковые объекты и регулярные выражения, исключения и обработка ошибок, делегаты и события. Вторая часть целиком посвящена созданию приложений и содержит обсуждение ADO.NET, ASP.NET и Windows Forms. Технология ASP.NET включает в себя как Web Forms для быстрой разработки веб-приложений, так и Web Services для создания объектов без графического интерфейса, оказывающих услуги в Интернете. В третьей части рассматривается платформа .NET Framework. Особое внимание уделено атрибутам и отражению, удаленным объектам, вычислительным потокам и синхронизации, а также потокам ввода/вывода. Здесь же представлены способы взаимодействия с объектами COM.

ISBN 5-93286-038-3
ISBN 0-596-00309-9 (англ)

© Издательство Символ-Плюс, 2003
Authorized translation of the English edition © 2002, 2001 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 25.11.2002. Формат 70x100 1/16 . Печать офсетная.
Объем 43 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 9

I. Язык программирования C# 19

1. C# и .NET Framework 21

Платформа .NET 21
.NET Framework 22
Компиляция и язык MSIL 25
Язык программирования C# 26

2. Начинаем. Программа Hello World 28

Классы, объекты и типы 28
Разработка программы Hello World 35
Использование отладчика Visual Studio .NET 39

3. Основы языка программирования C# 43

Типы 44
Переменные и константы 48
Выражения 55
Пробельные символы 55
Операторы 57
Операции 72
Пространства имен 80
Директивы препроцессора 82

4. Классы и объекты 86

Определение классов 87
Создание объектов 92
Статические члены класса 99
Уничтожение объектов 103
Передача параметров 106
Перегрузка методов и конструкторов 112
Инкапсуляция данных в свойствах 115
Поля, предназначенные только для чтения 118

5. Наследование и полиморфизм 120

Специализация и обобщение 120
Наследование 123
Полиморфизм 127
Абстрактные классы 133
Корень всех классов – класс Object 137
Упаковка и распаковка типов 139
Вложенные классы 141

6. Перегрузка операций 144

Ключевое слово operator 144
Поддержка других языков платформы .NET 145
Создание новых операций 146
Логические пары 146
Операция проверки на равенство 146
Операции преобразования типов 147

7. Структуры 154

Определение структур 155
Создание структур 157

8. Интерфейсы 162

Реализация интерфейса 163
Обращение к методам интерфейса 171
Переопределение реализации интерфейса 177
Явная реализация интерфейса 181

9. Массивы, индексаторы и классы коллекций 191

Массивы 191
Оператор foreach 196
Индексаторы 210
Интерфейсы классов коллекций 220
Класс ArrayList 225
Очереди 237
Стек 240
Словари 243

10. Строки и регулярные выражения 251

Строки 252
Регулярные выражения 267

11. Обработка исключений 279

Вызов и обработка исключений 280
Объекты Exception 289

Вызов пользовательских исключений 293
Повторный вызов исключения 295

12. Делегаты и события 299

Делегаты 300
События 320

II. Программирование на C# 331

13. Создание Windows-приложений 333

Создание простой формы Windows 335
Создание приложения Windows Forms 347
Документирующие комментарии XML 371
Развертывание приложения 372

14. Доступ к данным с помощью ADO.NET 384

Реляционные базы данных и язык SQL 385
Объектная модель ADO.NET 389
Приступаем к работе с моделью ADO.NET 390
Использование управляемых поставщиков OLE DB 394
Использование элементов управления с привязкой данных 397
Изменение записей в базе данных 407
Модель ADO.NET и технология XML 424

15. Создание веб-приложений с помощью Web Forms 426

Среда Web Forms 427
Создание веб-формы 431
Добавление элементов управления 434
Привязка данных 437
Реакция на отправляющие события 445
Технология ASP.NET и язык C# 447

16. Веб-службы 448

SOAP, WSDL и Discovery 449
Построение веб-службы 450
Создание класса-посредника 457

III. CLR и .NET Framework 461

17. Сборки и контроль версий 463

PE-файлы 463
Метаданные 464
Границы безопасности 464
Контроль версий 464
Манифесты 464

Многомодульные сборки 466
Закрытые сборки 474
Совместно используемые сборки 476

18. Атрибуты и отражение 483

Атрибуты 483
Стандартные атрибуты 484
Пользовательские атрибуты 486
Отражение 490
Динамическая генерация кода 500

19. Маршалинг и удаленные компоненты 525

Домены приложений 527
Контекст 537
Удаленные объекты 540

20. Потоки и синхронизация 551

Потоки 552
Синхронизация 561
Состояние гонки и взаимные блокировки 573

21. Потоки данных 575

Файлы и каталоги 576
Чтение и запись данных 588
Асинхронный ввод/вывод 594
Сетевой ввод/вывод 600
Веб-потоки 618
Сериализация 621
Изолированная память 630

22. Взаимодействие .NET и COM 634

Импорт элементов управления ActiveX 634
Импорт компонентов COM 643
Экспорт компонентов .NET 651
Техника P/Invoke 653
Указатели 656

Приложение. Ключевые слова языка C# 661

Алфавитный указатель 665

Предисловие

Приблизительно каждые десять лет новый подход к программированию накатывается, подобно цунами. В начале 80-х новыми технологиями были операционная система Unix, работавшая на рабочих станциях, и новый мощный язык, разработанный в AT&T и названный C. Начало 90-х принесло с собой Windows и C++. Каждая из этих разработок представляла принципиальное изменение подхода к программированию. Пришло время следующей волны – появилась платформа .NET и язык C#. И цель данной книги – помочь читателю оседлать эту волну.

Фирма Microsoft «все поставила на карту» под названием .NET. Когда компания такого размера и с таким влиянием тратит миллиарды долларов на реорганизацию своей структуры для поддержки новой платформы, программисты не могут оставить это без внимания. Оказывается, .NET коренным образом меняет способ мышления программиста. Если коротко, она представляет собой новую платформу разработки программного продукта, предназначенную для упрощения объектно-ориентированного программирования в Интернете. В качестве основного языка этой объектно-ориентированной платформы, нацеленной на Интернет, выбран C#. Его создатели хорошо усвоили уроки, преподанные такими языками, как C (высокая производительность), C++ (объектно-ориентированная структура), Java (сборка мусора, высокая безопасность) и Visual Basic (быстрая разработка). В результате получился язык, идеально подходящий для разработки компонентных n-уровневых распределенных веб-приложений.

Об этой книге

Данная книга является учебником как по языку C#, так и по программированию приложений .NET с его помощью. Читателям, знающим какой-либо язык программирования, достаточно будет бегло ознакомиться с содержимым первых глав, однако главу 1 прочитать все-таки необходимо, поскольку она представляет обзор языка и платформы .NET. Новичку же стоит прочитать эту книгу так, как Король Червей рекомендовал Белому Кролику: «Начни с начала, продолжай, пока не дойдешь до конца. Как дойдешь – кончай!»[1]

---

[1] Льюис Кэррол «Приключения Алисы в Стране чудес», пер. Н. Демуровой.

Как организована эта книга

Часть I посвящена тонкостям языка C#. В части II говорится, как создавать приложения .NET, а в части III описано, как использовать C# с библиотекой времени выполнения CLR (Common Language Runtime) платформы .NET.

Часть I. Язык программирования C#

Глава 1 «C# и .NET Framework» является введением в язык C# и платформу .NET Framework.

Глава 2 «Начинаем. Программа Hello World» демонстрирует простую программу, создающую контекст для последующего обсуждения, знакомит читателя с инструментальной средой разработки Visual Studio .NET и с некоторыми концепциями языка C#.

Глава 3 «Основы языка программирования C#» представляет основы языка, от базовых типов данных до ключевых слов.

Классы определяют новые типы и позволяют программисту расширить язык так, чтобы лучше смоделировать решаемую задачу. В главе 4 «Классы и объекты» описаны компоненты, являющиеся сердцем и душой языка C#.

Классы могут быть сложными представлениями и абстракциями реальных вещей. В главе 5 «Наследование и полиморфизм» обсуждается, как классы взаимодействуют друг с другом и в каких отношениях они могут находиться.

Глава 6 «Перегрузка операций» учит читателя добавлять операции в типы, определенные пользователем.

Главы 7 и 8 представляют «Структуры» и «Интерфейсы», соответственно. И те и другие являются близкими родственниками классов. Структуры – это облегченные объекты, обладающие меньшими возможностями, чем классы, и менее требовательные к операционной системе и объему памяти. Интерфейсы являются своего рода соглашениями. Они описывают, как работает класс, чтобы другие программисты могли взаимодействовать с объектами четко определенным способом.

Объектно-ориентированные программы нередко создают большое количество объектов. Эти объекты удобно объединять в группы и манипулировать ими как единым целым. Язык C# предоставляет широкую поддержку классов коллекций. В главе 9 «Массивы, индексаторы и классы коллекции» исследуются классы коллекций, предоставляемые библиотекой классов платформы (FCL, Framework Class Library), и демонстрируется, как можно создавать собственные типы коллекций.

В главе 10 «Строки и регулярные выражения» обсуждается, как в языке C# обрабатывается текст. Большинство веб- и Windows-приложений взаимодействует с пользователем, и строки играют важнейшую роль в пользовательском интерфейсе.

В главе 11 «Обработка исключений» поясняется, что делать с исключениями, представляющими собой объектно-ориентированный механизм обработки различных нештатных ситуаций.

Приложения, работающие как в Windows, так и во Всемирной паутине, управляются событиями. В языке C# события являются полноценными элементами языка. В главе 12 «Делегаты и события» описаны принципы работы с событиями и использование для этого *делегатов*, представляющих собой объектно-ориентированный механизм обратного вызова, гарантирующий безопасность преобразования типов.

Часть II. Приемы программирования на C#

Вторая и третья части книги будут интересны всем читателям, независимо от их опыта программирования на других языках. В этих главах исследуется сама платформа .NET.

Часть II посвящена тому, как писать программы для .NET, а именно Windows-приложения с помощью Windows Forms и веб-приложения с помощью Web Forms. Кроме того, в части II показано, как взаимодействовать с базами данных и создавать веб-службы.

На верхнем уровне этой инфраструктуры находится высокоуровневая абстракция операционной системы, предназначенная для облегчения разработки объектно-ориентированного программного продукта. В этот верхний уровень входят ASP.NET и Windows Forms. Технология ASP.NET включает в себя как набор инструментов Web Forms для быстрой разработки веб-приложений, так и веб-службы для создания веб-объектов без пользовательского интерфейса.

C# предоставляет модель быстрой разработки приложений (RAD, Rapid Application Development), аналогичную той, что ранее была доступна только в среде Visual Basic. В главе 13 «Создание Windows-приложений» описано, как пользоваться моделью RAD для создания профессиональных Windows-приложений с помощью среды разработки Windows Forms.

Независимо от того, предназначено приложение для Windows или Всемирной паутины, ему нередко приходится обрабатывать большие объемы информации. В главе 14 «Доступ к данным с помощью ADO.NET» обсуждается уровень ADO.NET платформы .NET Framework и описываются способы взаимодействия с сервером Microsoft SQL Server и другими поставщиками данных.

В главе 15 технология RAD, описанная в главе 13, объединяется с технологиями обработки данных, описанными в главе 14 с целью продемонстрировать «Создание веб-приложений с помощью Web Forms».

Не все приложения имеют пользовательский интерфейс. В главе 16 «Веб-службы» внимание уделено второй части технологии ASP.NET. *Веб-служба* – это распределенное приложение, предоставляющее опре-

деленные функциональные услуги с использованием веб-протоколов, как правило, протоколов XML и HTTP.

Часть III. CLR и .NET Framework

Среда времени выполнения – это окружение, в котором исполняются программы. CLR (Common Language Runtime, общеязыковая среда выполнения) – это сердце .NET. Она включает в себя систему приведения типов данных, которая действует на всей платформе и является общей для всех языков, входящих в .NET. Среда CLR несет ответственность за управление памятью и подсчет ссылок на объекты.

Другой ключевой особенностью среды .NET CLR является *сборка мусора*. В отличие от традиционного программирования на C/C++, разработчик, пишущий на C#, не отвечает за уничтожение объектов. Остались в прошлом долгие часы, потраченные на поиски утечки памяти; среда CLR «прибирает» за программистом, когда тот или иной объект становится ненужным. Сборщик мусора CLR ищет в куче объекты, на которые нет ссылок, и освобождает занимаемую ими память.

Платформа .NET и библиотека классов расширяется в платформе среднего уровня, где реализована инфраструктура, включающая в себя классы поддержки, в том числе типы, обеспечивающие связь между процессами, обработку XML, работу с потоками, ввод/вывод, безопасность, диагностику и т. д. В средний уровень входят также компоненты доступа к базам данных, известные под общим именем ADO.NET и обсуждаемые в главе 14.

В третьей части книги рассматриваются взаимоотношения языка C# со средой выполнения CLR и библиотекой классов платформы FCL.

В главе 17 «Сборки и контроль версий» проводится различие между частными (закрытыми) и разделяемыми сборками и демонстрируется, как нужно создавать сборки и управлять ими. В терминологии .NET *сборкой* называется коллекция файлов, представленная пользователю в виде единой библиотеки динамической компоновки (DLL) или выполняемого файла. Сборка является базовой единицей для повторного использования кода, контроля версий, защиты и развертывания.

Сборки платформы .NET содержат обширную информацию о классах, методах, свойствах, событиях и т. п., представленную в виде метаданных. Эти метаданные компилируются в программу и извлекаются из нее использующим ее приложением с помощью отражения. В главе 18 «Атрибуты и отражение» показано, как добавлять метаданные в программу, как создавать пользовательские атрибуты и как читать эти метаданные с помощью отражения. Кроме того, глава содержит обсуждение динамических вызовов, приема программирования, при котором методы вызываются с помощью позднего связывания (то есть на этапе выполнения), а заканчивается глава демонстрацией *динамичес-*

кой генерации кода, нетривиальной техники построения самомодифицирующегося кода.

Платформа .NET Framework была разработана специально для поддержки веб-приложений и распределенных приложений. Компоненты, созданные на языке C#, могут находиться в разных процессах на одном компьютере или на разных компьютерах в локальной сети или Интернете. *Маршалинг* (*marshaling*) представляет собой способ взаимодействия с отсутствующими в данном процессе объектами, а *отдаление* (*remoting*) – технологию связи с такими объектами. Эта тема освещается в главе 19 «Маршалинг и удаленные компоненты».

Библиотеки классов платформы предоставляют широкую поддержку асинхронного ввода/вывода, а также включают в себя другие классы, делающие явную манипуляцию вычислительными потоками необязательной. Тем не менее, в языке C# есть средства управления как вычислительными потоками, так и синхронизацией, обсуждаемые в главе 20 «Потоки и синхронизация».

В главе 21 «Потоки данных» рассматриваются потоки данных, механизм, который позволяет не только взаимодействовать с пользователем, но и получать данные из Интернета. В главе подробно описывается *сериализация* (*serialization*) – технология, позволяющая записать объектный граф на диск и прочитать его с диска.

В главе 22 «Взаимодействие .NET и COM» исследуется взаимодействие с компонентами COM, созданными вне управляемой среды платформы .NET Framework. Существует возможность вызова из кода C# компонентов COM, а также выззова из COM компонентов, созданных на языке C#.

Книга завершается приложением, где перечисляются ключевые слова языка C#.

Для кого эта книга

Эта книга написана для программистов, которые хотят разрабатывать приложения на платформе .NET. Без сомнения, многие из них уже имеют опыт работы с такими языками, как C++, Java или Visual Basic (VB). Возможно, некоторые читатели программируют на других языках, а иные вообще не являются профессиональными программистами, но работали с HTML и другими веб-технологиями. Эта книга ориентирована на все названные категории читателей, хотя людям, никогда не писавшим программы, некоторые разделы покажутся трудными.

Языки C# и Visual Basic .NET

Все языки платформы .NET Framework равны. Однако, перефразируя Джорджа Оруэлла, можно сказать, что некоторые языки равнее дру-

гих. C# является великолепным языком для разработок на платформе .NET. Он исключительно гибок, устойчив и хорошо продуман создателями. В настоящее время он чаще других языков используется в статьях и книгах по .NET-программированию.

Вполне возможно, что некоторые программисты, работающие на VB, предпочтут изучение C# совершенствованию своих навыков в языке VB на уровне VB.NET. Это будет естественным выбором, поскольку переход от VB6 к VB.NET вряд ли окажется легче, чем от VB6 к C# (по крайней мере, с точки зрения автора). Кроме того, исторически сложилось, что программирование на языках семейства C оплачивается выше программирования на языках семейства Basic, хотя, возможно, это и не вполне справедливо. На практике программисты, работающие на VB, никогда не пользовались таким уважением и материальными благами, которых заслуживают, и язык C# предлагает им прекрасную возможность перейти на более высокооплачиваемую работу.

Как бы то ни было, приглашаются все, кто имеет опыт программирования на VB! Автор книги имел в виду и эту категорию читателей и постарался облегчить им переход на другой язык.

Языки C# и Java

Возможно, программисты, пишущие на языке Java, смотрят на C# со смешанным чувством тревоги, радости и обиды. Предполагалось, что C# будет в определенном смысле «оторван» от языка Java. Автор воздержится от комментариев по поводу религиозной войны между Microsoft и «всеми, кто не Microsoft». Однако объективности ради следует признать, что C# многому научился от Java. Но язык Java сам многому научился от C++, а тот позаимствовал синтаксис от C, который, в свою очередь, извлек уроки из опыта других языков. Все мы стоим на плечах гигантов.

Переход на C# для программистов, привыкших к Java, не составит труда – они найдут в нем сходный синтаксис и знакомую и комфортную семантику. Возможно, программисты, работающие на Java, хотят, чтобы особое внимание было сосредоточено на различиях между двумя языками, что позволило бы им использовать C# максимально эффективно. Автор постарался удовлетворить это желание, делая комментарии по ходу изложения (см. замечания, предназначенные для программистов, пишущих на Java).

Языки C# и C++

Хотя программировать в среде .NET на языке C++ можно, это непросто и неестественно. Честно говоря, автор этих строк, программировавший на языке C++ десять лет и написавший дюжину книг по данному предмету, предпочел бы визит к стоматологу программированию на

языке C++ в среде .NET. Возможно, причина тому в более дружественном поведении C#. Как бы то ни было, узнав язык C#, автор не хочет возвращаться к старому.

Впрочем, программисту, привыкшему к C++, следует быть очень внимательным. На его пути встретится немало ловушек, и автор постарался отметить их, расставив в тексте книги предупреждения специально для профессионалов в области C++.

Соглашения, используемые в этой книге

В оформлении книги приняты следующие соглашения:

Курсив используется для обозначения:

- путей в файловой системе, имен файлов и названий программ;
- адресов в Интернете, таких как имена доменов и URL-адреса;
- новых терминов в местах, где даются их определения.

`Моноширинный шрифт` используется для обозначения:

- командных строк и параметров, которые следует ввести в компьютер буквально;
- имен и ключевых слов в примерах программ, включая имена методов, переменных и классов.

`Моноширинный наклонный шрифт` обозначает замещаемые элементы синтаксиса или кода, например изменяемые или необязательные конструкции.

`Моноширинный полужирный шрифт` используется для выделения участков кода.

Особое внимание уделяйте замечаниям, выделенным в тексте следующим образом:

Это совет. Он содержит дополнительную информацию по обсуждаемой теме.

Это предупреждение. Оно помогает справиться с проблемами и даже избежать их.

Техническая поддержка

В числе своих обязанностей автор видит поддержку написанных им книг на веб-сайте:

http://www.LibertyAssociates.com

Там же можно найти исходный код всех примеров, приводимых в этой книге, дискуссионную группу и отдельный раздел с ответами на воп-

росы относительно C#. Однако прежде чем посылать свой вопрос, просмотрите FAQ (перечень наиболее часто задаваемых вопросов) и список замеченных опечаток (errata). Если это не помогло решить проблему, посылайте вопрос в дискуссионный центр.

Самый эффективный способ получить помощь – задать вопрос, сформулированный максимально точно, а лучше написать маленькую программу, которая иллюстрировала бы возникшую проблему. В качестве альтернативы читатель может посетить различные группы новостей или дискуссионные центры в Интернете. Фирма Microsoft ведет целый ряд групп новостей, а Developmentor (*http://www.develop.com*) поддерживает замечательный дискуссионный лист сообщений, пришедших по электронной почте на тему .NET. То же самое можно сказать и про сайт Чарльза Кэррола (Charles Carroll), находящийся по адресу *http://www.asplists.com*.

Обратная связь

Коллектив издательства максимально тщательно проверил информацию, содержащуюся в этой книге, однако не исключено, что некоторые сведения устарели (или даже ошибочны!). Письма с указанием замеченных ошибок и предложениями относительно следующих изданий направляйте, пожалуйста, по адресу:

O'Reilly & Associates, Inc.
005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (телефон в США или Канаде)
(707) 829-0515 (международный/местный телефон)
(707) 829-0104 (факс)

На сайте издательства O'Reilly & Associates есть веб-страница, посвященная этой книге, где находятся примеры, список ошибок и планы на переиздание. Вся эта информация доступна по адресу в Интернете:

http://www.oreilly.com/catalog/progcsharp2

Адрес для замечаний и технических вопросов по книге:

bookquestions@oreilly.com

Более подробную информацию об этой и других книгах, а также технические статьи и дискуссию на тему C# и .NET Framework можно найти на сайте издательства O'Reilly & Associates:

http://www.oreilly.com

и на сайте O'Reilly .NET DevCenter:

http://www.oreillynet.com/dotnet

Благодарности

Чтобы обеспечить точность и полноту изложения и гарантировать, что книга будет соответствовать нуждам и интересам профессиональных программистов, я заручился поддержкой таких замечательных программистов, как Дональд Кси (Donald Xie), Дэн Хэрвиц (Dan Hurwitz), Сет Вайс (Seth Weiss), Сью Линч (Sue Lynch), Клифф Джеральд (Cliff Gerald) и Том Пэтр (Tom Petr). Джим Кулберт (Jim Culbert) не только написал рецензию на книгу и внес ряд ценных предложений, но и постоянно указывал мне на нужды программистов-практиков. Трудно переоценить вклад Джима в эту книгу.

Майк Вудринг (Mike Woodring) из Developmentor за неделю сообщил мне больше сведений о CLR, чем удалось бы узнать за полгода самостоятельных изысканий. Победить двух страшных зверей, C# и .NET, мне помогли многие сотрудники Microsoft и O'Reilly, в том числе Эрик Гуннерсон (Eric Gunnerson), Роб Ховард (Rob Howard), Пит Обермейер (Piet Obermeyer), Джонатан Хокинс (Jonathan Hawkins), Питер Дрейтон (Peter Drayton), Брэд Меррил (Brad Merrill), Бен Альбахари (Ben Albahari) и другие. Наверное, самый удивительный программист из всех, кого я знаю, – это Сьюзен Уоррен (Susan Warren). Я глубоко благодарен ей за помощь и советы.

Я нахожусь в неоплатном долгу перед Джоном Осборном (John Osborn), представившим меня издательству O'Reilly. Валери Куэрсиа (Valerie Quercia), Брайан Мак-Дональд (Brian McDonald), Джефф Холком (Jeff Holcomb), Клэр Клутье (Claire Cloutier) и Татьяна Диаз (Tatiana Diaz) сделали книгу гораздо лучше, чем представленная рукопись. Роб Романо (Rob Romano) создал некоторые иллюстрации и улучшил другие.

Многие читатели написали мне, указав на опечатки и небольшие ошибки в первом издании. Их усилия не были потрачены зря. Мы особенно признательны Sol Bick, Brian Cassel, Steve Charbonneau, Randy Eastwood, Andy Gaskall, Bob Kline, Jason Mauss, Mark Phillips, Christian Rodriguez, David Solum, Erwing Steininger, Steve Thomson, Greg Torrance и Ted Volk. Все мы хорошо потрудились над исправлением всех ошибок во втором издании. Еще мы подчистили книгу, чтобы убедиться, что в ней не появилось новых неточностей и что весь код компилируется и запускается правильно с последним релизом Visual Studio .NET. Но если вы все же найдете ошибку, проверьте, пожалуйста, список errata на моем сайте (*http://www.LibertyAssociates.com*) и, если эта ошибка новая, сообщите мне о ней по электронной почте *jliberty@libertyassociates.com*.

Ну и наконец, особая благодарность Брайану Джепсону (Brian Jepson), который отвечал и за повышенное качество второго издания, и за его своевременность. Я очень высоко оценил затраченные им усилия.

I

Язык программирования C#

1

C# и .NET Framework

Целью разработчиков C# было создание простого, безопасного, современного, объектно-ориентированного, нацеленного на Интернет, высокопроизводительного языка для работы на платформе .NET. C# – новый язык, но в его основу положен опыт последних трех десятилетий. Подобно тому как в детях можно найти черты их родителей и даже бабушек и дедушек, в C# легко обнаружить влияние таких языков, как Java, C++, Visual Basic и некоторых других.

Основная тема этой книги – язык C# и его применение в качестве инструмента программирования на платформе .NET. В своих предыдущих работах[1] по языку C++ я отстаивал ту точку зрения, что начинать изучение языка следует без ориентации на Windows или Unix. В случае C# такой подход не имел бы смысла. Этот язык нужно изучать специально для того, чтобы создавать .NET-приложения, в противном случае сама цель языка была бы утрачена. Таким образом, в этой книге язык рассматривается не в вакууме, а исключительно в контексте платформы Microsoft .NET и применительно к разработке приложений для локальных систем и Интернета.

Данная глава знакомит читателя как с языком C#, так и с платформой .NET, в частности с .NET Framework.

Платформа .NET

Когда в июле 2000 года корпорация Microsoft анонсировала язык C#, эта акция была частью более значительного события – объявления о

[1] Д. Либерти «Освой самостоятельно C++ за 21 день» – Киев: Вильямс, 2000 г.

платформе .NET. По сути своей платформа .NET является новой структурой создания программного продукта, которая предоставляет прикладной программный интерфейс (API) к своим службам, а также API-интерфейсы классических операционных систем Windows (особенно семейства Windows 2000). Это достигается за счет объединения ранее разрозненных технологий, созданных Microsoft в конце 90-х. Среди них можно назвать службы компонентов COM+, структуру разработки веб-приложений ASP, поддержку XML и объектно-ориентированного дизайна, поддержку новых протоколов веб-служб, таких как SOAP, WSDL и UDDI, а также нацеленность на Интернет. Все это интегрировано в единую архитектуру DNA.

Корпорация Microsoft утверждает, что до 80% средств, направленных на исследования и разработки, тратится на платформу .NET и связанные с ней технологии. Результаты такой политики на сегодняшний день выглядят впечатляюще. Например, область охвата платформы .NET просто огромна. Платформа состоит из четырех групп программных продуктов.

- Набор языков, куда входят C# и Visual Basic .NET; набор инструментальных средств разработки, в том числе Visual Studio .NET; обширная библиотека классов для построения веб-служб и приложений, работающих в Windows и в Интернете; а также среда выполнения программ CLR (Common Language Runtime, общеязыковая среда выполнения), в которой выполняются объекты, построенные на этой платформе.
- Набор серверов .NET Enterprise Servers, ранее известных под именами SQL Server 2000, Exchange 2000, BizTalk 2000 и др., которые предоставляют специализированные функциональные возможности для обращения к реляционным базам данных, использования электронной почты, оказания коммерческих услуг «бизнес-бизнес» (B2B) и т. д.
- Богатый выбор коммерческих веб-служб, называемых .Net My Services. За умеренную плату разработчики могут пользоваться этими службами при построении приложений, требующих идентификации личности пользователя и других данных.
- Новые некомпьютерные устройства, поддерживающие средства .NET, от сотовых телефонов до игровых приставок.

.NET Framework

Microsoft .NET поддерживает не только языковую независимость, но и языковую интеграцию. Это означает, что разработчик может наследовать от классов, обрабатывать исключения и использовать преимущества полиморфизма при одновременной работе с несколькими языками. Платформа .NET Framework предоставляет такую возможность

с помощью спецификации *Common Type System* (CTS, общая система типов), которой должны следовать все компоненты .NET. Например, в .NET любая сущность является объектом какого-нибудь класса, производного от корневого класса `System.Object`. Спецификация CTS поддерживает такие общие понятия, как классы, делегаты (с поддержкой обратных вызовов), ссылочные и размерные типы.

Кроме того, .NET включает спецификацию *Common Language Specification* (CLS, общая языковая спецификация), которая устанавливает основные правила языковой интеграции. Спецификация CLS определяет минимальные требования, предъявляемые к языку платформы .NET. Компиляторы, удовлетворяющие этой спецификации, создают объекты, способные взаимодействовать друг с другом. Любой язык, соответствующий требованиям CLS, может использовать все возможности библиотеки FCL (Framework Class Library, библиотека классов платформы).

Платформа .NET Framework является надстройкой над операционной системой, в качестве которой может выступать любая версия Windows[1], и состоит из ряда компонентов. На сегодняшний день платформа .NET Framework включает в себя:

- Четыре официальных языка: C#, VB.NET, Managed C++ (управляемый C++) и JScript .NET.
- Объектно-ориентированную среду CLR, совместно используемую этими языками для создания приложений под Windows и для Интернета.
- Ряд связанных между собой библиотек классов под общим именем FCL.

Архитектурные компоненты платформы .NET Framework представлены на рис. 1.1.

Самым важным компонентом платформы .NET Framework является CLR, предоставляющая среду, в которой выполняются программы. Она включает в себя виртуальную машину, во многих отношениях аналогичную виртуальной машине Java. На верхнем уровне среда активизирует объекты, производит проверку безопасности, размещает объекты в памяти, выполняет их, а также запускает сборщик мусора. (CTS также является частью CLR.)

На рис. 1.1 над уровнем CLR находится набор базовых классов платформы, а над ним расположены слой классов данных и XML, а также слой классов для создания веб-служб (Web Services), веб- и Windows-приложений (Web Forms и Windows Forms). Собранные воедино, эти классы известны под общим именем FCL (Framework Class Library). Это одна из самых больших библиотек классов в истории программи-

[1] Благодаря архитектуре среды CLR в качестве операционной системы может выступать любая версия Unix и вообще любая ОС.

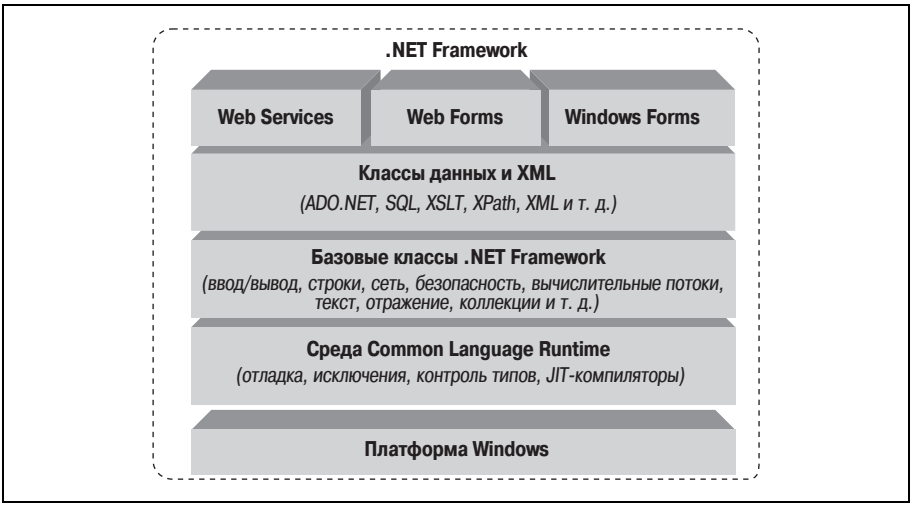

Рис. 1.1. *Архитектура .NET Framework*

рования. Она предоставляет объектно-ориентированный API-интерфейс ко всем функциональным возможностям, инкапсулированным платформой .NET. Имея в своем составе более 4000 классов, библиотека FCL способствует быстрой разработке настольных, клиент-серверных и других приложений и веб-служб.

Набор базовых классов платформы – нижний уровень FCL – аналогичен набору классов Java. Эти классы предоставляют элементарную поддержку ввода-вывода, манипуляций со строками, управления безопасностью, сетевой связи, управления вычислительными потоками, манипуляций с текстом, работы с отражениями и коллекциями и т. д.

Над этим уровнем находится уровень классов, которые расширяют базовые классы с целью обеспечения управления данными и XML. Классы данных позволяют реализовать управление информацией, хранящейся в серверных базах данных. В число этих классов входят классы SQL (Structured Query Language, язык структурированных запросов), дающие программисту возможность обращаться к долговременным хранилищам данных через стандартный интерфейс SQL. Кроме того, набор классов, называемый ADO.NET, позволяет оперировать постоянными данными. Платформа .NET Framework поддерживает также целый ряд классов, позволяющих манипулировать XML-данными и выполнять поиск и преобразования XML.

Базовые классы, классы данных и XML расширяются классами, предназначенными для построения приложений на основе трех различных технологий: Web Services (веб-службы), Web Forms (веб-формы) и Windows Forms (Windows-формы). Веб-службы включают в себя ряд классов, поддерживающих разработку облегченных распределяемых компонентов, которые могут работать даже с брандмауэрами и про-

граммами трансляции сетевых адресов (NAT). Поскольку веб-службы применяют в качестве базовых протоколов связи стандартные протоколы HTTP и SOAP, эти компоненты поддерживают в киберпространстве подход «plug-and-play».

Инструментальные средства Web Forms и Windows Forms позволяют применять технику Rapid Application Development (RAD, быстрая разработка приложений) для построения веб- и Windows-приложений. Эта техника сводится к перетаскиванию элементов управления с панели инструментов на форму, двойному щелчку по элементу и написанию кода, обрабатывающего события, связанные с этим элементом.

Более подробное описание платформы .NET Framework можно найти в книге Туана Тая (Thuan Thai) и Хонга К. Лэма (Hoang Q. Lam) «.NET Framework Essentials» издательства O'Reilly & Associates, 2001 г.[1]

Компиляция и язык MSIL

На платформе .NET программы компилируются не в исполняемые файлы, а в файлы MSIL, выполняемые средой CLR. (MSIL является аббревиатурой *Microsoft Intermediate Language*, промежуточный язык Microsoft.) Файлы MSIL (сокращенно IL), генерируемые компилятором C#, *идентичны* IL-файлам, генерируемым компиляторами с других языков .NET. В этом смысле платформа остается в неведении относительно языка. Самой важной характеристикой среды CLR является то, что она *общая*; одна среда выполняет как программы, написанные на C#, так и программы на языке VB.NET.

Исходный код на языке C# компилируется в IL-код во время построения проекта. Код IL сохраняется в файле на диске. Во время выполнения программы IL-код снова компилируется *JIT-компилятором* (*Just In Time compiler*); этот процесс часто называется *JIT-компиляцией* (*JIT'ing*). В результате генерируется машинный код, выполняемый процессором.

Стандартный JIT-компилятор работает *по запросу*. Когда вызывается тот или иной метод, JIT-компилятор анализирует IL-код и производит высокоэффективный машинный код, выполняемый очень быстро. JIT-компилятор достаточно интеллектуален, чтобы распознать, был ли код уже скомпилирован, поэтому во время выполнения программы компиляция происходит лишь при необходимости. По ходу своего выполнения .NET-программа работает все быстрее и быстрее, так как повторно используется уже скомпилированный код.

Спецификация CLS подразумевает, что все языки платформы .NET генерируют очень похожий IL-код. В результате объекты, созданные на

[1] Туан Тай, Хонг К. Лэм «Платформа .NET. Основы», 2 издание – СПб: Символ-Плюс, 2002 г.

одном языке, доступны и могут наследоваться на другом. То есть можно создать базовый класс на языке VB.NET, а производный от него класс – на языке C#.

Язык программирования C#

Язык C# обезоруживает своей простотой – в нем насчитывается около 80 ключевых слов и десяток встроенных типов данных. Тем не менее, он оказывается исключительно выразительным, когда дело доходит до реализации современных концепций программирования. Язык C# включает в себя самую полную поддержку структурного, компонентно-ориентированного и объектно-ориентированного программирования, которую только можно ожидать от современного языка, «стоящего на плечах» C++ и Java.

Язык C# был создан в фирме Microsoft небольшой командой, возглавляемой двумя выдающимися разработчиками, Андерсом Хейльсбергом (Anders Hejlsberg) и Скоттом Вильтамутом (Scott Wiltamuth). Хейльсберг известен как создатель языка Turbo Pascal, популярного среди разработчиков программ для PC, а также как руководитель коллектива, создавшего Borland Delphi, первую удачную интегрированную среду разработки программ типа «клиент-сервер».

В основе любого объектно-ориентированного языка лежит поддержка определения классов и работы с ними. Классы определяют новые типы, позволяющие программисту расширить язык так, чтобы наиболее удачно смоделировать решаемую задачу. Синтаксис языка C# содержит ключевые слова для объявления новых классов и их методов и свойств, а также для реализации инкапсуляции, наследования и полиморфизма – трех столпов объектно-ориентированного программирования.

В языке C# все, что относится к объявлению класса, находится в самом объявлении. Объявлению класса не требуются ни файлы заголовков, ни файлы IDL (Interface Definition Language, язык определения интерфейса). Более того, C# поддерживает новый XML-стиль встроенного документирования, что существенно упрощает создание встроенных комментариев и вывод справочной документации по приложению.

Кроме того, язык C# поддерживает *интерфейсы* (*interfaces*) – средство заключения контракта с классом на предоставление услуг, оговоренных интерфейсом. В языке C# класс может наследоваться только от одного родителя, но в нем могут быть реализованы несколько интерфейсов. Реализуя интерфейс, класс обещает предоставить функциональность, описанную интерфейсом.

C# также обеспечивает поддержку *структур* (*structs*), причем значение этого понятия сильно изменилось по сравнению с C++. В языке C# структура представляет собой ограниченный, упрощенный тип. Когда

создается его экземпляр, он предъявляет меньшие требования к операционной системе и запрашивает меньше памяти, чем обычный класс. Структура не может наследоваться от класса, и от нее нельзя наследовать классы, однако структура в состоянии реализовать интерфейс.

C# обладает такой объектно-ориентированной функциональностью, как свойства, события и декларативные конструкции (называемые *атрибутами*). Компонентно-ориентированное программирование поддерживается средой CLR в том смысле, что метаданные хранятся вместе с кодом класса. Метаданные описывают класс, включая его методы, свойства, требования безопасности и другие атрибуты, например возможность сериализации, а код описывает логику выполнения функций. Таким образом, скомпилированный класс является самодостаточной единицей. Если среда исполнения знает, как прочитать метаданные и код, ей не нужна никакая другая информация, чтобы пользоваться классом. Имея в своем распоряжении язык C# и среду CLR, программист может добавлять к классу произвольные метаданные, снабжая его соответствующими атрибутами. Аналогичным образом он может читать метаданные класса с помощью типов из среды CLR, поддерживающих отражение.

Сборка (*assembly*) – это коллекция файлов, которая предстает перед программистом в виде единой библиотеки динамической компоновки (DLL) или исполняемого файла (EXE). В технологии .NET сборка является базовой единицей для повторного использования, контроля версий, защиты и развертывания. Среда CLR предоставляет программисту ряд классов, позволяющих манипулировать сборками.

Последнее, что стоит отметить, говоря о C#, – он позволяет программисту непосредственно обращаться к участкам памяти с помощью указателей в стиле C++ и ключевых слов, помечающих подобные действия как небезопасные. Кроме того, программист может предупреждать сборщик мусора, что объекты, на которые ссылается указатель, нельзя удалять, пока они не будут освобождены.

2

Начинаем. Программа Hello World

Традиция начинать книги по программированию с программы Hello World освящена временем. В этой главе мы создадим, скомпилируем и выполним простенькую программу Hello World на языке C#. Анализ этой короткой программы позволит обсудить важнейшие особенности языка.

Пример 2.1 иллюстрирует фундаментальные элементы простой программы на языке C#.

Пример 2.1. Простая программа Hello World на C#

```
class HelloWorld
{
    static void Main()
    {
        // Использовать объект "системная консоль"
        System.Console.WriteLine("Hello World");
    }
}
```

В результате компиляции и выполнения программы Hello World на консоли появятся слова «Hello World». Рассмотрим эту программу поближе.

Классы, объекты и типы

Суть объектно-ориентированного программирования состоит в создании новых типов. *Тип* представляет собой какое-то понятие. Оно может быть абстрактным, например таблица данных или вычислительный поток, а может быть чуть более «осязаемым», как кнопка в окне. Тип определяет общие свойства и поведение представляемого понятия.

Если в программе используются три экземпляра объектов типа «кнопка в окне», скажем кнопки OK, Cancel и Help, у каждого экземпляра будут свои свойства и особенности поведения. Например, все они имеют размер (возможно, у каждого свой), позицию на экране (естественно, их позиции различаются), надпись (например, «OK», «Cancel» и «Help»). Аналогичным образом они имеют сходное поведение, то есть их можно вывести на экран, активизировать, щелкнуть и т. д. Таким образом, детали могут различаться, но все кнопки имеют один и тот же *тип*.

Как и во многих других языках объектно-ориентированного программирования, в C# тип определяется *классом*, а отдельные экземпляры класса называются *объектами*. В следующих главах говорится, что помимо классов в C# есть и другие типы, например перечисления, структуры и делегаты, но сейчас все внимание сосредоточим на классах.

Программа Hello World объявляет один-единственный тип: класс `HelloWorld`. Тип в языке C# определяется следующим образом. Он объявляется как класс с помощью ключевого слова `class`, ему дается имя (в данном случае `HelloWorld`), а затем определяются его свойства и поведение. Определение свойств и поведения класса C# должно быть заключено в фигурные скобки (`{}`).

Вниманию программистов, пишущих на языке C++: Точка с запятой после закрывающей фигурной скобки не ставится.

Методы

Класс имеет как свойства, так и поведение. Поведение определяется методами класса; свойства обсуждаются в главе 3.

Метод (*method*) – это *функция*, которой обладает класс. Методы класса иногда так и называют *функциями класса*. Методы определяют, что конкретно может делать класс и как он себя ведет. Обычно методам даются имена, в которых отражаются их действия, например `WriteLine()` или `AddNumbers()`. Однако в данном случае методу класса дано специальное имя `Main()`. Оно не описывает действие, а сообщает среде CLR, что метод является главным, или первым, методом вашего класса.

В отличие от C++, имя `Main()` пишется в C# с большой буквы и может возвращать значения типа `int` или `void`.

CLR вызывает метод `Main()` при запуске программы. `Main()` является точкой входа в программу, и каждая программа на языке C# должна иметь метод `Main()`.[1]

---

[1] Технически возможно иметь несколько методов `Main()` в одной программе. В этом случае следует с помощью параметра командной строки `/main` сообщить компилятору C#, какой класс содержит метод `Main()`, являющийся точкой входа в программу.

Объявление метода представляет собой своего рода соглашение между его создателем и потребителем метода. Возможно, и даже очень вероятно, что создателем и потребителем будет один и тот же программист, но это совсем не обязательно. Бывает, что один член команды разработчиков создает метод, а другой применяет его.

Чтобы объявить метод, следует указать тип возвращаемого значения, за которым следует имя метода. Независимо от того, имеет ли метод аргументы, за именем метода должны следовать круглые скобки. Например, строка:

```
int myMethod(int size);
```

объявляет метод с именем `myMethod`, имеющий один целочисленный аргумент, который внутри метода будет называться `size`. Описанный здесь метод возвращает целочисленное значение. Тип возвращаемого значения сообщает потребителю метода, какого типа данные возвратит метод после завершения своей работы.

Некоторые методы не возвращают значения. О них говорят, что они возвращают *пустое* значение, которое специфицируется ключевым словом `void`. Например, строка:

```
void myVoidMethod();
```

объявляет метод, не возвращающий значение и не имеющий аргументов. В C# следует всегда указывать либо тип возвращаемого значения, либо `void`.

Комментарии

Программа на языке C# может содержать комментарии. Взгляните на первую строчку после открывающей фигурной скобки:

```
// Использовать объект "системная консоль"
```

Текст начинается с двух слэшей (//), которые обозначают *комментарий*. Комментарий адресован программисту и никак не влияет на работу программы. В языке C# существует три вида комментариев.

Первый только что рассмотрен. Два слэша сообщают, что текст справа от них вплоть до конца строчки является комментарием. Такой комментарий называется *комментарием в стиле C++* (однострочный комментарий).

Второй вид комментария, известный как *комментарий в стиле C* (многострочный комментарий), начинается с открывающего знака комментария (символы /*) и заканчивается закрывающим (*/). Такое обозначение позволяет растянуть комментарий на несколько строк, не предваряя каждую двумя слэшами (пример 2.2).

Пример 2.2. Пример многострочного комментария

```
class Hello
{
    static void Main()
    {
        /* Используйте объект "системная консоль",
           как показано в главе 2 */
        System.Console.WriteLine("Hello World");
    }
}
```

Комментарии в стиле C++ могут быть вложены в комментарии в стиле C. Поэтому принято использовать комментарий в стиле C++, где это возможно, а комментарием в стиле C «выключать» отдельные блоки кода.

Третий вид комментариев в C# применяется для связывания кода с XML-документом; он обсуждается в главе 13.

Консольные приложения

Hello World представляет собой пример *консольного* приложения. Оно не имеет пользовательского интерфейса, то есть списков, кнопок, окон и т. д. Текстовый ввод/вывод происходит через стандартную консоль (обычно это окно командной строки или окно DOS на экране компьютера). Тот факт, что обсуждение пока ограничивается консольными приложениями, позволяет упростить первые примеры в этой книге и сосредоточить внимание собственно на языке. В последующих главах будут рассмотрены как веб-, так и Windows-приложения, и тогда мы сосредоточимся на Visual Studio .NET – инструментальном средстве разработки пользовательского интерфейса.

В рассматриваемом примере метод `Main()` всего лишь выводит на монитор текст «Hello World». Вывод на монитор осуществляется объектом с именем `Console`. У этого объекта есть метод `WriteLine()`, который принимает в качестве аргумента *строку* (последовательность символов) и выводит ее в стандартный поток ввода/вывода. При запуске этой программы на компьютерном мониторе появляется окно, командное или DOS, в котором находится текст «Hello World».

Метод вызывается оператором принадлежности (.). Иными словами, чтобы вызвать метод `WriteLine()` объекта `Console`, следует написать `Console.WriteLine(...)`, поставив вместо многоточия выводимую строку.

Пространства имен

`Console` – это всего лишь один из огромного количества типов, входящих в состав .NET Framework Class Library (FCL) – библиотеки классов платформы .NET. У каждого класса есть имя, так что FCL содер-

жит тысячи имен, среди которых ArrayList, Hashtable, FileDialog, DataException, EventArgs и т. д. Сплошные имена – сотни, тысячи, десятки тысяч имен.

Это обстоятельство порождает некоторые трудности. Ни один разработчик не сможет запомнить все имена, существующие в структуре .NET, и рано или поздно программист создает объект с уже существующим именем. Что произойдет, если, скажем, читатель создаст класс Hashtable и обнаружит, что он конфликтует с классом Hashtable, предоставляемым платформой .NET? Ведь каждый класс C# должен обладать уникальным именем.

Конечно, читатель может переименовать свой Hashtable в mySpecialHashtable, но это проигрышный путь. Кто-нибудь другой создаст тип с таким именем, и жизнь обоих программистов превратится в сплошной кошмар.

Выход заключается в создании *пространства имен* (*namespace*). Пространство имен ограничивает область применения имени, делая его осмысленным только в рамках данного пространства.

Предположим, Джим работает инженером. Слово «инженер» имеет довольно широкое значение. Чем он конкретно занимается? Строит дома? Разрабатывает программное обеспечение? Конструирует локомотивы?

Можно уточнить его специальность, сказав, что Джим занимается наукой или железнодорожным транспортом. В языке C#, применяя английские слова, можно эти понятия обозначить как science.engineer или train.engineer.

Тогда слово science (наука) или train (железнодорожный транспорт) будет обозначать пространство имен, ограничивающее область применения слова, следующего за точкой, то есть engineer (инженер). Так создается область, в которой имя обладает смыслом.

Идем дальше. Джим занимается не наукой «вообще». Он закончил Массачусетский технологический институт с дипломом программиста. Тогда объект Jim можно определить более точно: science.software.engineer. В этой классификации предполагается, что пространство имен software (программное обеспечение) имеет смысл в пределах пространства имен science, а engineer в этом контексте имеет смысл в пространстве software. Если теперь читатель узнает, что профессия Шарлотты transportation.train.engineer, он не будет долго раздумывать, каким именно инженером она является. Два понятия engineer существуют параллельно, каждое в своем пространстве имен.

Аналогичным образом, зная, что в .NET есть класс Hashtable, принадлежащий пространству имен System.Collections, программист создаст класс Hashtable в пространстве имен ProgCSharp.DataStructures, и никакого конфликта не будет. Каждый класс существует в своем пространстве.

В примере 2.1 имя объекта Console ограничено пространством имен System:

```
System.Console.WriteLine()
```

Оператор принадлежности (.)

В примере 2.1 оператор принадлежности (.) используется двояко. Он предоставляет программисту метод (и данные) класса (в рассмотренном примере метод WriteLine()), а также ограничивает имя класса определенным пространством имен (Console в рамках System). Такой подход срабатывает, поскольку в обоих случаях выполняется «движение вниз» к искомому предмету. На верхнем уровне находится пространство имен System (которое содержит все объекты System, представляемые платформой); тип Console существует в пределах этого пространства, а метод WriteLine() — функция класса в типе Console.

Во многих случаях пространства имен делятся на подпространства. Например, System содержит ряд подпространств имен, таких как Configuration, Collections, Data и т. д., причем подпространство Collections само разбито на множество подпространств.

Пространства имен помогают программисту в организации и систематизации создаваемых типов. Написав сложную программу на языке C#, ее разработчик может создать собственную иерархическую структуру пространств имен, и нет никаких ограничений на ее глубину. Вообще пространства имен задумывались для реализации принципа «разделяй и властвуй» по отношению к иерархии объектов.

Ключевое слово using

Вместо того чтобы постоянно указывать слово System перед Console, достаточно сообщить, что в программе будут использованы типы из пространства имен System. Это делается оператором:

```
using System;
```

в самом начале кода, как показано в примере 2.3.

Пример 2.3. Ключевое слово using

```
using System;
class Hello
{
    static void Main()
    {
        // Console из пространства имен System
        Console.WriteLine("Hello World");
    }
}
```

Обратите внимание, что оператор `using System` расположен до определения класса `HelloWorld`.

Хотя программист может заявить об использовании пространства имен `System`, в C# (в отличие от некоторых других языков) нельзя заранее сообщить об использовании объекта `System.Console`. Код, приведенный в примере 2.4, компилироваться не будет.

Пример 2.4. Код, который не будет откомпилирован (недопустим в C#)

```
using System.Console;
class Hello
{
    static void Main()
    {
        // Console из пространства имен System
        WriteLine("Hello World");
    }
}
```

Компилятор выдаст следующее сообщение об ошибке:

```
error CS0138: A using namespace directive can only be applied to namespaces;
'System.Console' is a class not a namespace

(ошибка CS0138: Директива использования пространства имен может быть
применима только к пространству имен;
'System.Console' является классом, а не пространством имен)
```

Конструкция `using` позволяет значительно сократить объем набираемого кода, но она может свести на нет преимущества пространства имен, засоряя пространство большим количеством трудноразличимых имен. Общепринятым решением этой проблемы является применение ключевого слова `using` для встроенных пространств имен и пространств имен вашей фирмы, но не для компонентов сторонних производителей.

Чувствительность к регистру символов

Язык C# чувствителен к регистру символов. Это означает, что имена `writeLine`, `WriteLine` и `WRITELINE` разные. К сожалению, в отличие от Visual Basic (VB), среда разработки C# не обнаруживает ошибки, связанные с регистром символов. Написав одно и то же имя дважды в разных регистрах, программист закладывает в свою программу ошибку, которую будет нелегко распознать.

Чтобы предотвратить возникновение подобных ошибок и сберечь время и силы, следует придерживаться определенной системы именования переменных, функций, констант и т. д. В этой книге принято соглашение называть переменные в соответствии с нотацией «верблюд» (например, `someVariableName`), а функции, константы и свойства – в соответствии с нотацией Pascal (например, `SomeFunction`).

Единственным различием между этими двумя нотациями является регистр самой первой буквы. В нотации Pascal имена начинаются с большой буквы.

Ключевое слово static

Метод `Main()` в примере 2.1 имеет одну важную особенность. Перед объявлением возвращаемого типа `void` (который, как помнит читатель, означает, что метод не возвращает значение) стоит ключевое слово `static`:

```
static void Main()
```

Ключевое слово `static` показывает, что метод `Main()` можно вызывать, не создавая объект типа `HelloWorld`. Этот нетривиальный вопрос будет подробно освещен в следующих главах. Одной из проблем, возникающих при изучении нового языка программирования, является необходимость применять какие-нибудь сложные конструкции до того, как они станут окончательно понятными. Отнеситесь пока к объявлению метода `Main()` как к магическому заклинанию.

Разработка программы Hello World

Есть минимум два способа ввести, откомпилировать и выполнить программы, приведенные в этой книге: с помощью интегрированной среды разработки Visual Studio .NET или с помощью текстового редактора и компилятора (а также некоторых инструментальных средств, описанных далее в этой книге), запускаемых из командной строки.

Хотя читатель *может* разрабатывать программы и вне среды Visual Studio .NET, интегрированная среда разработки (Integrated Development Environment, IDE) обладает целым рядом достоинств. Сюда входят автоматический контроль отступов, система завершения слов Intellisense, подсветка ключевых слов, справочная система. И самое главное – среда IDE имеет в своем составе мощный отладчик и много других инструментов.

Хотя в этой книге предполагается, что читатель работает в среде Visual Studio .NET, изложение строится вокруг языка и платформы, а не инструментальных средств. Читатель может скопировать любой пример в текстовый редактор, например Notepad или Emacs, сохранить его как текстовый файл и указать его имя в командной строке при вызове компилятора C#, поставляемого в комплекте .NET Framework SDK. Обратите внимание, что некоторые примеры в последних главах книги создают Windows- и веб-формы с помощью Visual Studio .NET. Тем не менее, даже их можно вручную набрать в редакторе Notepad, если вы предпочитаете самый трудный путь.

Редактирование приложения Hello World

Чтобы создать программу Hello World в среде IDE, выберите значок Visual Studio .NET в меню Start (Пуск) или на рабочем столе, а затем выберите в меню File → New → Project. Появится окно New Project (рис. 2.1). При первом запуске Visual Studio это окно может появиться и без подсказок.

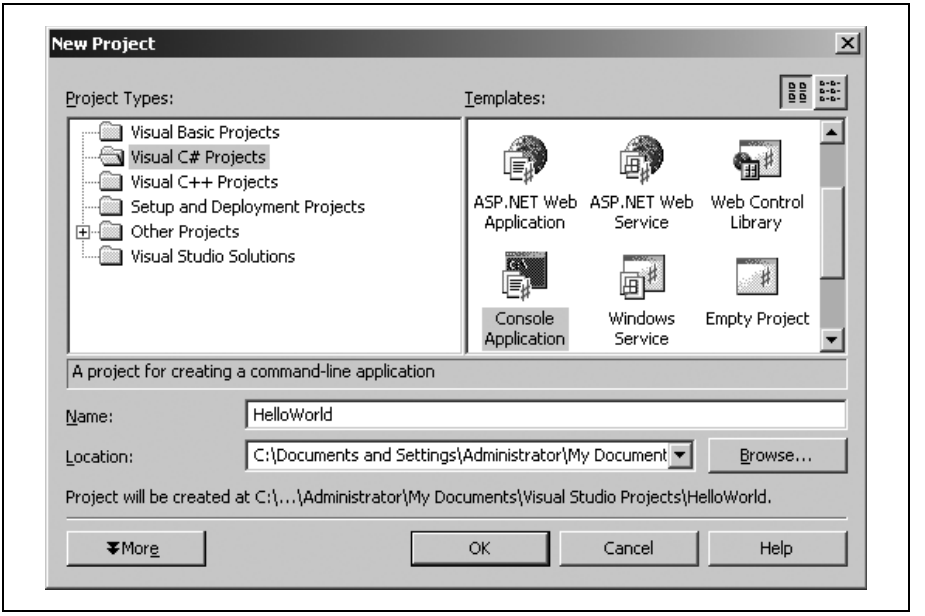

Рис. 2.1. *Создание консольного приложения C# в Visual Studio .NET*

Чтобы открыть приложение, выберите Visual C# Projects в окне Project Types, а затем выберите Console Application в окне Templates. Теперь можно вводить имя проекта и выбирать каталог для хранения файлов. Щелкните кнопку OK. Появится новое окно, в которое можно ввести код из примера 2.1 (рис. 2.2).

Обратите внимание, что Visual Studio .NET создает пространство имен, беря за основу название проекта (`HelloWorld`), и автоматически добавляет в код оператор `using System`, поскольку почти каждой программе нужны типы из пространства имен `System`.

Visual Studio .NET создает класс с именем `Class1`, но его можно переименовать. Дав классу другое имя, не забудьте переименовать и файл *Class1.cs*. Например, чтобы воспроизвести на своем компьютере пример 2.1, читатель должен будет изменить имя `Class1` на `HelloWorld`, а файл *Class1.cs*, указанный в окне Solution Explorer, переименовать в *HelloWorld.cs*.

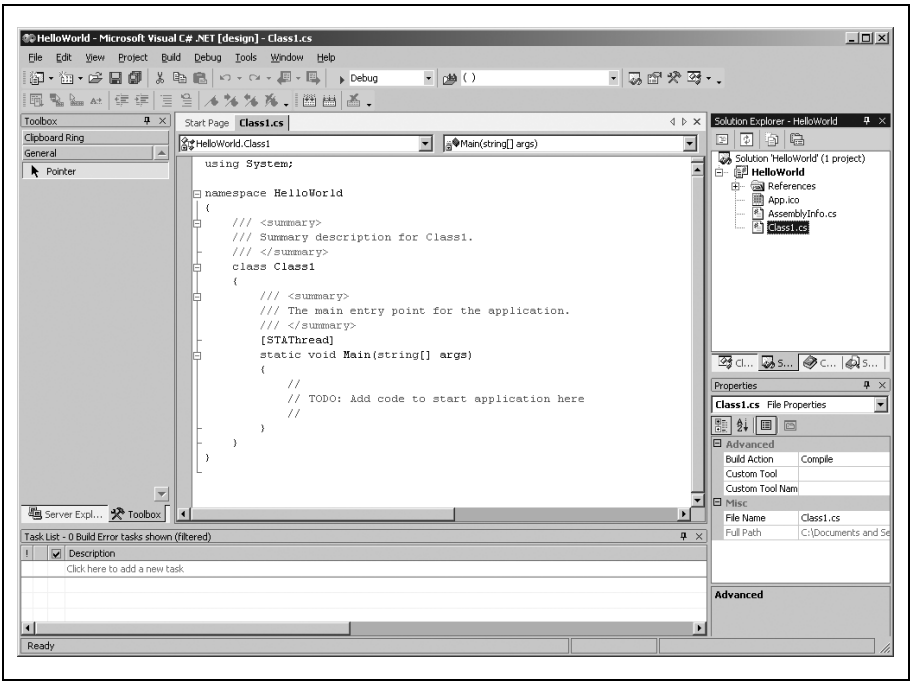

Рис. 2.2. *Текстовый редактор, открытый для нового проекта*

Наконец, Visual Studio .NET создает скелет программы с комментарием `TODO`, чтобы было с чего начать. Создавая программу из примера 2.1, удалите из скелета метода `Main()` аргументы (`string[] args`) и комментарий, затем скопируйте в тело метода следующие две строчки:

```
// Использовать объект "системная консоль"
System.Console.WriteLine("Hello World");
```

Читатель, работающий вне среды Visual Studio .NET, должен открыть текстовый редактор, набрать код примера 2.1 и сохранить файл под именем *Hello.cs*.

Компиляция и запуск приложения Hello World

Существует несколько способов откомпилировать и выполнить программу Hello World из среды Visual Studio .NET. Обычно программист выполняет каждую задачу отдельно, выбирая команды в меню, щелкая кнопки или зачастую пользуясь комбинациями клавиш.

Например, чтобы откомпилировать программу Hello World, можно нажать комбинацию клавиш <Ctrl>+<Shift>+<B> или выбрать в меню пункт Build → Build Solution. В качестве альтернативы можно щелкнуть по кнопке Build, расположенной на одноименной панели инструментов (возможно, потребуется щелкнуть правой кнопкой по любой панели

инструментов, чтобы добавить панель инструментов Build). Значок этой кнопки изображен на рис. 2.3.

Чтобы выполнить программу Hello World без отладчика, можно нажать комбинацию <Ctrl>+<F5> на клавиатуре, выбрать в меню пункт Debug → Start Without Debugging или щелкнуть по кнопке Start Without Debugging на панели инструментов Build (вам может потребоваться дополнительная настройка, чтобы сделать эту панель видимой). Значок этой кнопки изображен на рис. 2.4. Программу можно выполнить, не выделяя компиляцию в отдельный этап. В зависимости от выбранных параметров (Tools → Options), среда IDE сохранит файл, откомпилирует и выполнит его, возможно, запрашивая подтверждение на выполнение каждого шага.

Рис. 2.3. *Значок кнопки Build*

Рис. 2.4. *Кнопка Start without debugging*

Автор настоятельно рекомендует читателю поэкспериментировать со средой разработки Visual Studio .NET. Это основной инструмент .NET-разработчика, и следует научиться пользоваться им. Время, потраченное на то, чтобы привыкнуть к Visual Studio .NET, многократно окупится в ближайшие месяцы. Отложите книгу и приступайте, а автор готов подождать.

Чтобы откомпилировать и выполнить программу Hello World при помощи компилятора C# с интерфейсом командной строки, проделайте следующие шаги:

1. Сохраните пример 2.1 в файле с именем *hello.cs*.
2. Откройте окно командной строки (выберите Start → Run (Пуск → Выполнить) и введите `cmd`).
3. Введите строку:

   ```
   csc /debug hello.cs
   ```

   На этом шаге будет построен выполняемый файл (EXE). Если программа содержит ошибки, компилятор сообщит о них в командном окне. Параметр командной строки `/debug` вставляет в скомпилированный код специальные символы, которые позволяют запускать exe-файл в отладчике и видеть номера строк при трассировке стека. (Трассировка стека будет отображена, когда программа сгенерирует не обрабатываемую вами ошибку.)
4. Чтобы выполнить программу, введите:

   ```
   Hello
   ```

В командном окне появятся священные слова «Hello World».

Just In Time компиляция

Скомпилировав *Hello.cs* с помощью *csc*, вы получите исполняемый (EXE) файл. Помните, однако, что этот *.exe* файл содержит коды команд промежуточного языка Microsoft Intermediate Language (MSIL), описанного ранее.

Если бы вы написали это приложение на VB.NET или любом другом языке, соответствующем спецификации .NET Common Language Specification, оно было бы скомпилировано в тот же самый MSIL. Код промежуточного языка IL, создаваемый различными языками, практически неотличим; это первый пунт общей языковой спецификации.

Помимо IL-кода (который по духу подобен байт-коду Java) компилятор создает доступный только для чтения раздел в *.exe*-файле, в который помещает стандартный исполняемый заголовок Win32. Компилятор указывает точку входа внутри этого раздела, и загрузчик операционной системы передает управление на эту точку входа, когда вы запускаете программу, так же как и для обычного приложения Windows.

Операционная система не может выполнить код IL, однако точка входа ничего не делает кроме передачи управления Just In Time компилятору .NET (описанному в главе 1). JIT-компилятор генерирует машинный код для центрального процессора, который вы можете найти в обычном файле *.exe*. Ключевой особенностью JIT-компилятора является то, что функции компилируются только тогда, когда они используются для выполнения, то есть как раз вовремя (just-in-time).

Использование отладчика Visual Studio .NET

Существует мнение (возможно, спорное), что единственным ценным инструментом в любой среде разработки является отладчик. Отладчик Visual Studio весьма мощный, так что не жалейте времени на его изучение. Основы процесса отладки сами по себе очень просты. Главное надо уметь:

- устанавливать точку останова и выполнять программу вплоть до этой точки;
- заходить внутрь методов и обходить их;
- проверять и изменять значения переменных, данных класса и т. п.

Автор не собирается переписывать документацию по отладчику, однако перечисленные навыки настолько важны, что здесь приводится очень краткий курс по обращению с отладчиком.

Управлять действиями отладчика можно по-разному – обычно программист выбирает пункты меню или щелкает по кнопкам. Простейший способ установить точку останова – щелкнуть по левому полю. Среда IDE отмечает точку останова красным кружком, как показано на рис. 2.5.

```
for (int i = 0;i < 3; i++)
{
    winArray[i].DrawWindow();
}
}
```

Рис. 2.5. *Точка останова*

Обсуждение работы отладчика немыслимо без конкретных примеров. Фрагмент программы, приведенный на этих страницах, взят из главы 5, поэтому не предполагается, что читатель разберется в нем (впрочем те, кто имеет опыт работы на языках C++ или Java, поймут, что в нем происходит).

Чтобы запустить отладчик, выберите Debug → Start в меню или просто нажмите клавишу <F5>. Программа будет откомпилирована и выполнена вплоть до точки останова. Когда программа остановится, желтая стрелка отметит оператор, который должен быть выполнен следующим (рис. 2.6).

```
for (int i = 0;i < 3; i++)
{
    winArray[i].DrawWindow();
}
}
```

Рис. 2.6. *Индикатор следующего оператора*

Когда программа достигла точки останова, можно проверить значения различных объектов. Например, можно выяснить значение переменной i, поместив на нее курсор и немного подождав (рис. 2.7).

```
for (int i = 0;i < 3; i++)
{          i=0
    winArray[i].DrawWindow();
}
```

Рис. 2.7. *Показ значения переменной*

Кроме того, отладчик среды IDE предоставляет программисту несколько окон, например окно Locals, где выведены значения всех локальных переменных (рис. 2.8).

Значения базового типа, например целочисленные значения, представлены обычным образом (как у переменной `i`), а рядом с объектами стоит значок «плюс» (+). Эти объекты можно развернуть и посмотреть значения их внутренних данных, как показано на рис. 2.9. Объекты и их внутренние данные более подробно обсуждаются в следующих главах.

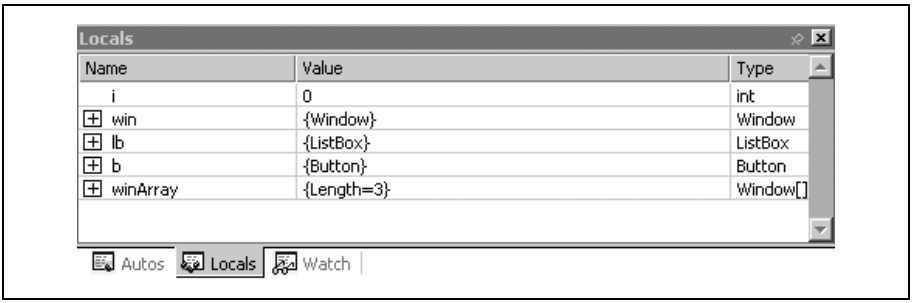

Рис. 2.8. *Окно Locals*

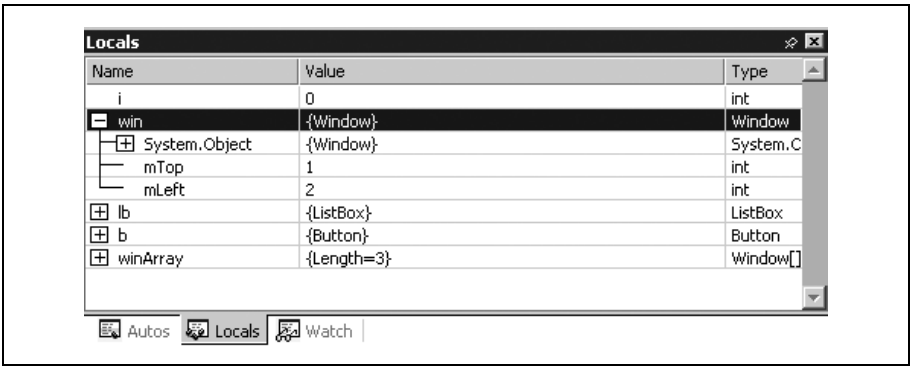

Рис. 2.9. *Развернутый объект в окне Locals*

Нажав клавишу <F11>, программист заходит внутрь метода. На рис. 2.10 показан заход внутрь метода `DrawWindow()` класса `WindowClass`.

Из рисунка понятно, что следующим выполняемым оператором в методе `DrawWindow()` будет вызов метода `WriteLine()`. Содержимое окна Autos обновилось и демонстрирует текущее состояние объектов.

Читателю предстоит еще многое узнать об отладчике, но это краткое введение поможет приступить к работе с ним. На многие вопросы, связанные с программированием, можно получить ответ, написав короткие демонстрационные программы и изучив их работу в отладчике. Хороший отладчик в определенном смысле является лучшим техническим средством обучения языку программирования.

```
    // simulates drawing the window
    public virtual void DrawWindow()
    {
        Console.WriteLine("Window: drawing Window at {0}, {1}",
            mTop, mLeft);
    }

   // these members are private and thus invisible
    // to derived class methods. We'll examine this
    // later in the chapter
    protected int mTop;
    protected int mLeft;

}

// ListBox derives from Window
public class ListBox : Window
{
    // constructor adds a parameter
```

Autos

Name	Value	Type
mLeft	2	int
mTop	1	int
this	{Window}	Window
System.Object	{Window}	System.Object
mTop	1	int
mLeft	2	int

Autos | Locals | Watch

Рис. 2.10. *Заход внутрь метода*

3

Основы языка программирования C#

В главе 2 была продемонстрирована очень простая программа на языке C#. Тем не менее, создание даже такой маленькой программы сопряжено с определенными трудностями, и некоторые существенные детали не обсуждались. В данной главе этим деталям уделено достаточно внимания благодаря подробному изучению синтаксиса и структуры языка C#.

В этой главе обсуждается система типов в C#, причем проводится различие между базовыми типами (`int`, `bool` и т. д.) и типами, определяемыми пользователем (типы, создаваемые как классы и интерфейсы). Кроме того, в настоящей главе излагаются основы программирования, например создание и использование переменных и констант. Затем вводятся такие понятия, как перечисления, строки, идентификаторы и операторы.

Вторая часть главы посвящена ветвлению программы, реализуемому операторами `if`, `switch`, `while`, `do...while`, `for` и `foreach`. Обсуждаются операции присваивания, логические, математические и операции отношения. Далее следует введение в пространства имен и краткое руководство по применению препроцессора C#.

Хотя язык C# принципиально предназначен для создания и обработки объектов, лучше всего начать с самых основных понятий, с помощью которых формируются объекты. Сюда входят базовые типы, являющиеся внутренней частью языка C#, и синтаксические элементы этого языка.

Типы

C# является языком со строгой типизацией. В таких языках программист должен объявлять тип всех создаваемых объектов (например, целых и дробных чисел, строк, окон, кнопок и т. д.), а компилятор предотвращает возникновение ошибок, следя за тем, чтобы объектам присваивались только данные разрешенного типа. Тип объекта сообщает компилятору о его размере (например, тип `int` показывает, что объект занимает 4 байта) и возможностях (например, кнопка может быть нарисована, нажата и т. д.).

Как и в языках C++ и Java, в C# типы делятся на две группы: *базовые* (*intrinsic*) типы, предлагаемые языком, и типы, *определяемые пользователем* (user-defined).

Кроме того, типы C# разбиваются на две другие категории: *размерные типы* (*value types*) и *ссылочные типы* (*reference types*).[1] Принципиальное различие между размерными и ссылочными типами состоит в способе хранения их значений в памяти. В первом случае фактическое значение хранится в стеке (или как часть большого объекта ссылочного типа). Адрес переменной ссылочного типа тоже хранится в стеке, но сам объект хранится в куче.

Если объект имеет большой размер, размещение его в куче дает ряд преимуществ. В главе 4 обсуждаются достоинства и недостатки ссылочных типов, а в данной главе все внимание сосредоточено на базовых типах, используемых в языке C#.

Дополнительно к упомянутым, язык C# поддерживает типы *указателей* (*pointers*), однако они используются редко и только с неуправляемым кодом. Неуправляемым называется код, созданный вне платформы .NET, например COM-объекты. Работа с COM-объектами описана в главе 22.

Базовые типы

Язык C# предлагает обычный набор базовых типов, который можно ожидать в современном языке программирования. Каждому из них соответствует тип, поддерживаемый общеязыковой спецификацией .NET (CLS). Соответствие базовых типов языка C# и типов платформы .NET гарантирует, что объекты, созданные в C#, могут быть использованы на равных основаниях с объектами, созданными в любом другом языке, удовлетворяющем требованиям .NET CLS (например, в языке VB.NET).

---

[1] Почти все базовые типы являются размерными типами. Исключение составляют типы `Object` (обсуждается в главе 5) и `String` (глава 10). Все пользовательские типы, кроме структур (глава 7), являются ссылочными.

У каждого типа свой неизменяемый размер. В отличие от C++, тип `int` в языке C# всегда занимает 4 байта, поскольку он отображается на тип `Int32` в спецификации .NET CLS. В табл. 3.1 приведены базовые типы значений, доступные в языке C#.

Таблица 3.1. Базовые типы значений

Тип	Размер (в байтах)	Тип .NET	Описание
`byte`	1	`Byte`	Байт без знака (значения от 0 до 255)
`char`	2	`Char`	Символы Unicode
`bool`	1	`Boolean`	`true` или `false`
`sbyte`	1	`SByte`	Байт со знаком (значения от -128 до 127)
`short`	2	`Int16`	Целое со знаком (короткое) (значения от -32 768 до 32 767)
`ushort`	2	`UInt16`	Целое без знака (короткое) (значения от 0 до 65 535)
`int`	4	`Int32`	Целое число со знаком от -2 147 483 647 до 2 147 483 647
`uint`	4	`UInt32`	Целое число без знака от 0 до 4 294 967 295
`float`	4	`Single`	Число с плавающей точкой. Содержит значения приблизительно от $+/-1.5*10^{-45}$ до $+/-3.4*10^{38}$ с семью значащими цифрами
`double`	8	`Double`	Число с плавающей точкой двойной точности. Содержит значения приблизительно от $+/-5.0*10^{-324}$ до $+/-1.7*10^{308}$ с 15–16 значащими цифрами
`decimal`	12	`Decimal`	Число до 28 знаков с фиксированным положением десятичной точки. Обычно используется в финансовых расчетах. Требует суффикса «m» или «M»
`long`	8	`Int64`	Целое число со знаком; значения от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
`ulong`	8	`UInt64`	Целое число без знака; значения от 0 до 0ffffffffffffffff

Вниманию программистов, пишущих на языках C и C++! Логические переменные могут принимать только значения `true` и `false`. Целочисленные значения не соответствуют логическим в C#, и неявное преобразование типов не производится.

В дополнение к перечисленным базовым типам в C# имеются еще два типа: `enum` (рассматриваемый далее в этой главе) и `struct` (см. главу 7). В главе 5 обсуждаются и другие тонкости, связанные с типами значе-

ний, например, рассказывается, как при помощи процесса, называемого *упаковкой* (*boxing*), заставить их вести себя словно ссылочные типы, а также о том, что типы значений не наследуются.

Стек и куча

Стек – это структура данных, используемая для хранения элементов по принципу «первым пришел – последним ушел» (аналог – стопка подносов в столовой самообслуживания). В данном случае под стеком понимается область памяти, обслуживаемая процессором, в которой хранятся локальные переменные.

В C# размерные типы (например, целые числа) размещаются в стеке. Под значение отводится отдельная область памяти, а ссылкой на нее является имя переменной.

Ссылочные типы (например, объекты) расположены в куче. Когда объект размещен в куче, возвращается его адрес, и этот адрес присваивается ссылке.

Сборщик мусора уничтожает объекты в стеке через некоторое время после того, как закончит существование фрейм стека, в котором они объявлены. В типичном случае фрейм стека определяется функцией. То есть если в пределах функции объявлена локальная переменная (как показано далее в этой главе), соответствующий объект будет помечен для сборки мусора по окончании функции.

Объект в куче подвергается сборке мусора через некоторое время после того, как уничтожена последняя ссылка на него.

Выбор базового типа

Обычно программист решает, какой тип целого (`short`, `int` или `long`) использовать, основываясь на диапазоне возможных значений. Например, переменная типа `ushort` может хранить значения от 0 до 65 535, зато `uint` – от 0 до 4 294 967 295.

Однако память – ресурс дешевый, а время программиста стоит очень дорого. Поэтому в большинстве случаев достаточно объявлять переменные типа `int`, если нет веских причин выбрать иной тип.

Типы со знаком выбираются программистами для хранения чисел в большинстве случаев, если нет веских причин воспользоваться беззнаковым типом.

Не следует поддаваться искушению удвоить диапазон положительных чисел, выбрав тип `ushort` вместо `short` (максимальное положительное число в этом случае будет 65 535 вместо 32 767). Проще и предпочтительнее воспользоваться целым со знаком (`int`), и тогда максимальное значение достигнет 2 147 483 647.

Если положительное значение является неотъемлемой характеристикой данных, следует применять целочисленные переменные без знака. Например, для хранения возраста человека лучше выбрать тип `uint`, поскольку возраст не может быть отрицательным.

Типы `float`, `double` и `decimal` предлагают программисту богатый выбор размера и точности данных. Для дробных чисел с небольшой точностью прекрасно подходит `float`. Обратите внимание: компилятор предполагает, что любое число с десятичной точкой имеет тип `double`, если не указано иначе. Чтобы назначить литералу тип `float`, поставьте после него букву `f`. (Присваивание литеральных значений подробно обсуждается далее в этой главе.)

```
float someFloat = 57f;
```

Тип `char` представляет символ в кодировке Unicode. Литералы этого типа являются либо простыми символами, либо символами Unicode, либо управляющей последовательностью, заключенными в апострофы. Например, `A` – это простой символ, а `\u0041` – символ Unicode. Управляющая последовательность – это специальные двухсимвольные лексемы, в которых первым символом является обратная наклонная черта. Например `\t` обозначает горизонтальную табуляцию. Общепринятые управляющие последовательности представлены в табл. 3.2.

Таблица 3.2. Общепринятые управляющие последовательности

Символ	Значение
`\'`	Апостроф
`\"`	Кавычки
`\\`	Обратный слэш
`\0`	Символ с нулевым кодом
`\a`	Сигнал
`\b`	Забой
`\f`	Перевод формата
`\n`	Новая строка
`\r`	Возврат каретки
`\t`	Горизонтальная табуляция
`\v`	Вертикальная табуляция

Преобразование базовых типов

Объекты одного типа могут быть преобразованы в объекты другого либо явно, либо неявно. Неявные преобразования выполняются автоматически; компилятор сам позаботится об этом. Явные преобразования происходят, когда программист переводит значение в другой тип. Семантика явного преобразования: «Эй! Компилятор! Я знаю, что де-

лаю!» Это все равно, что забивать гвозди огромным молотком: либо весьма эффективно, либо очень больно, в зависимости от того, по чему попадает молоток – по гвоздю или по пальцу.

Неявные преобразования происходят автоматически, причем сохранность данных гарантируется. Например, можно неявно преобразовать тип `short int` (2 байта) в `int` (4 байта). Какое бы значение ни хранилось в `short`, оно не потеряется во время преобразования в `int`:

```
short x = 5;
int y = x; // неявное преобразование
```

Конечно, преобразование в обратном направлении может привести к потере информации. Если значение типа `int` больше 32 767, оно будет усечено в ходе преобразования. Что касается неявного преобразования из `int` в `short`, компилятор не станет его выполнять:

```
short x;
int y = 500;
x = y;  // не компилируется
```

Явное преобразование необходимо выполнять с помощью соответствующего оператора:

```
short x;
int y = 500;
x = (short) y;  // OK
```

Для всех базовых типов определены правила преобразования. Иногда бывает удобно сформулировать аналогичные правила для типов, определенных пользователем. Это обсуждается в главе 5.

Переменные и константы

Переменная представляет собой типизированную область памяти. В предыдущих примерах *x* и *y* являются переменными. Переменным присваиваются значения, которые можно изменять программным образом.

Программист создает переменную, объявляя ее тип и указывая ее имя. При объявлении переменной ее можно инициализировать, а затем в любой момент ей можно присвоить новое значение, которое заменит собой предыдущее. Эти положения иллюстрируются примером 3.1.

Пример 3.1. Инициализация переменной и присваивание значения

```
class Values
{
   static void Main()
   {
      int myInt = 7;
```

```
        System.Console.WriteLine("Инициализация myInt: {0}",
           myInt);
        myInt = 5;
        System.Console.WriteLine("После присваивания myInt: {0}",
           myInt);
    }
}
```

Вывод:

```
Инициализация myInt: 7
После присваивания myInt: 5
```

Здесь переменная myInt инициализируется значением 7, это число выводится, переменной присваивается значение 5, и оно тоже выводится.

Метод WriteLine()

Платформа .Net Framework предоставляет полезный метод для вывода информации на экран. По ходу изложения читатель будет все больше узнавать об этом методе, но самое главное понятно уже сейчас. Метод System.Console.WriteLine() вызывается, как показано в примере 3.3. Ему передается строка, которая должна быть выведена на консоль (на экран), и, возможно, несколько необязательных параметров, которые будут замещены. В следующем коде:

```
System.Console.WriteLine("После присваивания myInt: {0}", myInt);
```

Будет напечатана строка «После присваивания myInt:», а за ней – значение переменной myInt. Позиция *подстановочного параметра* {0} указывает, где должно находиться значение первой выводимой переменной myInt. В данном случае это конец строки. Более подробно метод WriteLine() обсуждается в следующих главах.

Явное присваивание

В языке C# требуется, чтобы переменные были явно проинициализированы до их использования. Для проверки этого правила измените строчку, в которой инициализируется переменная myInt, на

```
int myInt;
```

и сохраните полученную программу (пример 3.2).

Пример 3.2. Использование неинициализированной переменной

```
class Values
{
   static void Main()
   {
      int myInt;
```

```
        System.Console.WriteLine
        ("Неинициализированная myInt: {0}",myInt);
        myInt = 5;
        System.Console.WriteLine("Присваивание myInt: {0}", myInt);
    }
}
```

При попытке скомпилировать этот пример компилятор C# выведет следующее сообщение:

```
3.1.cs(6,55): error CS0165: Use of unassigned local
variable 'myInt'
```

В языке C# нельзя пользоваться неинициализированной переменной; пример 3.2 некорректен.

Значит ли это, что надо инициализировать каждую переменную в программе? На самом деле – нет. Инициализировать переменную необязательно, однако программист должен присвоить ей значение до того, как воспользуется ею. В примере 3.3 представлена корректная программа.

Пример 3.3. Присваивание без инициализации

```
class Values
{
   static void Main()
   {
      int myInt;
      myInt = 7;
      System.Console.WriteLine("Присваивание myInt: {0}", myInt);
      myInt = 5;
      System.Console.WriteLine("Повторное присваивание myInt: {0}", myInt);
   }
}
```

Константы

Константа – это переменная, значение которой нельзя изменить. Переменные являются очень мощной конструкцией языка, но иногда удобно манипулировать значением, которое наверняка останется неизменным. Например, в программе, имитирующей химический эксперимент, могут понадобиться значения, соответствующие температуре кипения и замерзания воды. Программа будет понятнее, если переменные с этими значениями будут называться `FreezingPoint` (ТочкаЗамерзания) и `BoilingPoint` (ТочкаКипения), однако нельзя допустить, чтобы этим переменным присваивались другие значения. Как предотвратить присваивание? Ответ прост: воспользоваться константой.

Константы бывают трех видов: *литералы*, *символические константы* и *перечисления*. В операторе присваивания:

```
x = 32;
```

число 32 является литеральной константой. Его значение всегда равно 32. Нельзя присвоить другое значение числу 32, например, чтобы оно было равно 99; как ни старайтесь – ничего не выйдет.

Символические константы именуют постоянные значения. Программист определяет символьную константу с помощью ключевого слова const в соответствии со следующим синтаксисом:

```
const тип идентификатор = значение;
```

При объявлении константу надо инициализировать, и после этого ее невозможно изменить. Например:

```
const int FreezingPoint = 32;
```

Здесь 32 – это литеральная константа, а FreezingPoint – символическая, имеющая тип int. Практическое применение символических констант иллюстрируется примером 3.4.

Пример 3.4. Использование символических констант

```
class Values
{
   static void Main()
   {
      const int FreezingPoint = 32;   // градусы по Фаренгейту
      const int BoilingPoint = 212;

      System.Console.WriteLine("Точка замерзания воды: {0}",
            FreezingPoint );
      System.Console.WriteLine("Точка кипения воды: {0}",
            BoilingPoint );
      //BoilingPoint = 21;

   }
}
```

В примере 3.4 создаются две символические константы, FreezingPoint и BoilingPoint. Что касается стиля, то их имена написаны в соответствии с нотацией Pascal, однако в языке C# отсутствуют требования на этот счет.

Эти константы служат тем же целям, что и *литеральные значения* 32 и 212, то есть используются в выражениях, где требуется температура замерзания и кипения воды. Однако наличие имен делает их более осмысленными. Кроме того, если понадобится использовать в программе градусы по шкале Цельсия, будет достаточно инициализировать константы значениями 0 и 100 на этапе компиляции, а вся остальная часть кода останется прежней.

Чтобы убедиться, что константам нельзя присваивать значение, уберите знак комментария из последней строчки программы (выделена полужирным шрифтом). При компиляции такой версии будет выдана ошибка:

```
error CS0131: The left-hand side of an assignment must be
a variable, property or indexer
```

Перечисления

Перечисления (*enumerations*) являются мощной альтернативой константам. Перечисление – это особый размерный тип, состоящий из набора именованных констант (называемых *списком перечисления*).

В примере 3.4 были созданы две связанные по смыслу константы:

```
const int FreezingPoint = 32;
const int BoilingPoint = 212;
```

К ним можно добавить еще несколько:[1]

```
const int LightJacketWeather = 60;
const int SwimmingWeather = 72;
const int WickedCold = 0;
```

При таком определении констант программа выглядит несколько неуклюже, да и связь между ними не просматривается. *Перечисление*, предоставляемое языком C#, решает эти проблемы:

```
enum Temperatures
{
   WickedCold = 0,
   FreezingPoint = 32,
   LightJacketWeather = 60,
   SwimmingWeather = 72,
   BoilingPoint = 212,
}
```

Каждое перечисление имеет соответствующий тип. Это может быть любой целый тип (`int`, `short`, `long` и т. д.), кроме `char`. Синтаксис определения перечисления такой:

```
[атрибуты] [модификаторы] enum идентификатор
                    [:базовый-тип] {список-перечисления};
```

Атрибуты и модификаторы, являющиеся необязательными элементами этой конструкции, рассматриваются позже. В данный момент сосредоточим внимание на остальной части объявления. Оно начинается с ключевого слова `enum`, за которым следует идентификатор, например:

```
enum Temperatures
```

Базовый тип – это тип перечисления. Если опустить этот элемент (что обычно и делается), по умолчанию будет использован тип `int`. Однако

[1] Названия этих констант можно перевести как ПогодаДляЛегкойОдежды, ПогодаДляКупания и ЖуткийХолод; по шкале Цельсия температура соответствует примерно 16, 22 и –18 градусам. – *Примеч. пер.*

программист волен выбрать любой целый тип (например, ushort или long), кроме char. В следующем коде объявляется перечисление, состоящее из беззнаковых целых (uint):

```
enum ServingSizes :uint
{
    Small = 1,
    Regular = 2,
    Large = 3
}
```

Обратите внимание, что объявление перечисления включает в себя список. Список перечисления содержит конструкции, присваивающие константам значения, и эти конструкции разделяются запятыми.

Пример 3.5 представляет собой пример 3.4, переписанный с применением перечисления.

Пример 3.5. Использование перечисления для упрощения кода

```
class Values
{

   enum Temperatures
   {
      WickedCold = 0,
      FreezingPoint = 32,
      LightJacketWeather = 60,
      SwimmingWEather = 72,
      BoilingPoint = 212,
   }

   static void Main()
   {

      System.Console.WriteLine("Точка замерзания воды: {0}",
         Temperatures.FreezingPoint );
      System.Console.WriteLine("Точка кипения воды: {0}",
         Temperatures.BoilingPoint );

   }
}
```

Как нетрудно заметить, перечисление квалифицируется своим типом (например, Temperatures.WickedCold). По умолчанию значение перечисления отображается с использованием символического имени (например, BoilingPoint или FreezingPoint). Если необходимо отобразить значение константы перечисления, следует явно привести ее к базовому типу (int). Тогда в метод WriteLine() будет передано целочисленное значение и оно же будет выведено.

Каждой константе в перечислении соответствует числовое значение, в данном примере целое. Если значения не указаны явно, то перечисление начинается с нуля, а каждое следующее значение на 1 превышает предыдущее.

Так, если объявить перечисление:

```
enum SomeValues
{
   First,
   Second,
   Third = 20,
   Fourth
}
```

то значением константы `First` будет `0`, `Second` — `1`, `Third` — `20`, а `Fourth` — `21`.

Перечисления являются формальными типами, следовательно, для перевода их в целый тип требуется явное преобразование.

Вниманию программистов, пишущих на языке C++!
В языке C# перечисления несколько отличаются от перечислений C++, где запрещено присваивание целого значения перечислению, но разрешено преобразование перечислимого типа в целочисленный для присваивания его значения целочисленной переменной.

Строки

Практически невозможно написать программу на языке C# и обойтись без строк. Строковый объект содержит последовательность символов.

Программист объявляет строковую переменную с помощью ключевого слова `string`, аналогично созданию любого другого объекта:

```
string myString;
```

Строковый литерал создается заключением последовательности символов в кавычки:

```
"Hello World"
```

Распространенной практикой является инициализация строковой переменной строковым литералом:

```
string myString = "Hello World";
```

Более подробно строки рассматриваются в главе 10.

Идентификаторы

Идентификаторы – это имена, которые программисты дают типам, методам, переменным, константам, объектам и т. п. Идентификатор должен начинаться с буквы или символа подчеркивания.

Фирма Microsoft предлагает пользоваться *нотацией camel* для имен переменных (первая буква в нижнем регистре, например, `someName`) и

нотацией Pascal для имен методов и большинства других идентификаторов (первая буква в верхнем регистре, например SomeOtherName).

Фирма Microsoft больше не рекомендует венгерскую нотацию (например, iSomeInteger) и подчеркивания (например, SOME_VALUE).

Идентификаторы не должны вступать в конфликт с ключевыми словами. То есть нельзя создать переменную с именем int или class. Кроме того, идентификаторы чувствительны к регистру символов. Так, в C# имена myVariable и MyVariable считаются разными.

Выражения

Конструкции языка, в которых вычисляется значение, называются *выражениями* (*expressions*). Читатель может удивиться, как много операторов подпадает под это определение. Например, присваивание:

```
myVariable = 57;
```

является выражением, поскольку вычисляет присваиваемое значение, в данном случае 57.

Обратите внимание, что в приведенном выражении значение 57 присваивается переменной myVariable. Знак «равно» (=) говорит не о проверке на равенство, а о том, что правая часть (57) должна быть присвоена левой части (myVariable). Все операции языка C# (включая присваивание и равенство) обсуждаются далее в этой главе (см. раздел «Операции»).

Поскольку конструкция myVariable = 57 является выражением, возвращающим 57, ее можно использовать в составе другого оператора присваивания, например:

```
mySecondVariable = myVariable = 57;
```

В этом операторе происходит следующее. Литеральное значение 57 присваивается переменной myVariable. Значение этого присваивания (57) затем присваивается другой переменной, mySecondVariable. Короче говоря, значение 57 присваивается обеим переменным. Таким способом можно в одном операторе присвоить некоторое значение любому количеству переменных.

```
a = b = c = d = e = 20;
```

Пробельные символы

В языке C# такие символы, как пробел, табуляция и перевод строки, называются пробельными (поскольку на экране и при печати они вы-

глядят, как пробел). В операторах C# лишние пробельные символы (называемые таже символами-разделителями) обычно игнорируются. То есть можно написать:

```
myVariable = 5;
```

или:

```
myVariable    =                                5;
```

и компилятор будет считать эти операторы идентичными.

Исключением из этого правила являются пробельные символы в составе строки; они не игнорируются. Если написать код:

```
Console.WriteLine("Hello World")
```

то каждый пробел между словами «Hello» и «World» будет трактоваться как отдельный символ данной строки.

В большинстве случаев применение пробельных символов интуитивно понятно. Их предназначение – улучшить читаемость программы с точки зрения программиста; что касается компилятора, то ему все равно.

Тем не менее бывают ситуации, в которых пробельные символы начинают играть существенную роль. В то время как выражение:

```
int x  =  5;
```

неотличимо (для компилятора) от:

```
int x=5;
```

оно имеет мало общего с выражением:

```
intx=5;
```

Компилятор знает, что пробельный символ слева или справа от знака присваивания необязателен, но пробел между объявлением типа `int` и именем переменной `x` *абсолютно* необходим. И это неудивительно; пробельный символ позволяет компилятору правильно распознать ключевое слово `int` и отличить его от неизвестной конструкции `intx`. Программист волен вставлять между `int` и `x` сколько угодно символов-разделителей, однако хоть один такой символ должен быть непременно (обычно это пробел или табуляция).

Вниманию программистов, пишущих на языке Visual Basic! В C# символ конца строки не имеет специального значения. Операторы заканчиваются точкой с запятой, а не переводом строки. Символ продолжения строки отсутствует, поскольку в нем нет необходимости.

Операторы

В языке C# завершенная инструкция программы называется *оператором* (*statement*). Программа состоит из последовательности операторов C#, каждый из которых заканчивается точкой с запятой (;). Например:

```
int x;       // оператор
x = 23;      // еще один оператор
int y = x;   // и еще оператор
```

Операторы C# выполняются в естественном порядке. Компилятор начинает с первого из них и идет до последнего. Если бы не было переходов, программы были бы очень простыми и чрезвычайно ограниченными. В программе на языке C# возможны два вида переходов: *безусловные* и *условные*.

На ход выполнения программы влияют также операторы цикла и итерации, которые обозначаются ключевыми словами `for`, `while`, `do`, `in` и `foreach`. Циклы обсуждаются далее в этой главе. В данный момент сосредоточимся на основных способах реализации условного и безусловного перехода.

Операторы безусловного перехода

Безусловный переход реализуется одним из двух способов. Первый – это вызов метода. Когда компилятор встречает имя метода, он прекращает выполнение текущего метода и переходит к выполнению только что вызванного. Когда этот метод возвратит значение, выполнение вызвавшего метода возобновится со строчки, следующей за вызовом. Это иллюстрирует пример 3.6.

Пример 3.6. Вызов метода

```
using System;
class Functions
{
    static void Main()
    {
        Console.WriteLine("В Main! Вызов SomeMethod()...");
        SomeMethod();
        Console.WriteLine("Вернулись в Main().");

    }
    static void SomeMethod()
    {
        Console.WriteLine("Привет из SomeMethod!");
    }
}
```

Вывод:

```
В Main! Вызов SomeMethod()...
```

```
Привет из SomeMethod!
Вернулись в Main().
```

Выполнение программы начинается с метода `Main()` и продолжается вплоть до вызова `SomeMethod()` (вызов метода иногда называют активизацией). В этой точке программа переходит к указанному методу. Когда он закончится, выполнение программы будет возобновлено со строчки, следующей за вызовом.

Вторым способом реализации безусловного перехода является применение одного из следующих ключевых слов: `goto`, `break`, `continue`, `return` и `throw`. Более подробная информация о первых четырех операторах приведена далее в этой главе в разделах «Оператор switch: альтернатива вложенным if», «Оператор goto» и «Операторы continue и break». Пятый оператор, `throw`, обсуждается в главе 11.

Операторы условного перехода

Условный переход реализуется условным оператором, который обозначается ключевыми словами `if`, `else` или `switch`. Условный переход происходит только при выполнении определенного условия.

Операторы if...else

Операторы `if...else` осуществляют переход по условию. Условие – это выражение, проверяемое в заголовке оператора `if`. Если оно имеет значение `true`, выполняется оператор (или блок операторов) в теле оператора `if`.

Оператор `if` может сопровождаться необязательным оператором `else`. Оператор `else` выполняется, только если условие в заголовке оператора `if` имеет значение `false`:

```
if (выражение)
   оператор1
[else
   оператор2]
```

Примерно такое описание оператора `if` приводится в документации по компилятору. Здесь показано, что в заголовке оператора `if` стоит логическое *выражение* (выражение, результатом которого является *true* или *false*), заключенное в круглые скобки. Если его значение равно `true`, выполняется конструкция *оператор1*, которая может быть как одиночным оператором, так и блоком операторов в фигурных скобках.

Квадратные скобки в этом описании свидетельствуют о необязательности оператора `else`. Хотя подобное описание дает представление о синтаксисе оператора `if`, изучение примера 3.7 еще больше прояснит ситуацию.

Пример 3.7. Операторы if...else

```
using System;
class Values
{
   static void Main()
   {
      int valueOne = 10;
      int valueTwo = 20;

      if ( valueOne > valueTwo )
      {
         Console.WriteLine(
           "ValueOne: {0} больше, чем ValueTwo: {1}",
                valueOne, valueTwo);
      }
      else
      {
         Console.WriteLine(
          "ValueTwo: {0} больше, чем ValueOne: {1}",
                valueTwo,valueOne);
      }

      valueOne = 30; // увеличить значение valueOne

      if ( valueOne > valueTwo )
      {
         valueTwo = valueOne++;
         Console.WriteLine("\nПрисвоили valueTwo значение valueOne, ");
         Console.WriteLine("и увеличили ValueOne.\n");
            Console.WriteLine("ValueOne: {0}  ValueTwo: {1}",
                valueOne, valueTwo);
      }
      else
      {
         valueOne = valueTwo;
         Console.WriteLine("Сделали их равными. ");
            Console.WriteLine("ValueOne: {0}  ValueTwo: {1}",
                valueOne, valueTwo);
      }
   }
}
```

В примере 3.7 первый оператор `if` проверяет, превышает ли значение `valueOne` значение `valueTwo`. Операции сравнения «больше» (>), «меньше» (<) и «равно» (==) интуитивно понятны.

Проверка дает в результате `false` (поскольку `valueOne` равно 10, `valueTwo` равно 20 и, следовательно, `valueOne` *не* больше, чем `valueTwo`). Выполняется оператор `else`, который выводит:

```
ValueTwo: 20 больше чем ValueOne: 10
```

Условие второго оператора `if` имеет значение `true`, и все операторы в блоке `if` выполняются. В результате выводятся следующие строчки:

```
Присвоили valueTwo значение ValueOne,
и увеличили ValueOne.

ValueOne: 31  ValueTwo: 30
```

Блоки операторов

В любом месте, где компилятор C# предполагает встретить оператор, может находиться блок операторов. *Блок операторов* – это их последовательность, заключенная в фигурные скобки.

Таким образом, везде, где разрешается написать:

```
if (someCondition)
    someStatement;
```

можно с равным успехом написать:

```
if(someCondition)
{
    statementOne;
    statementTwo;
    statementThree;
}
```

Вложенные операторы if

В языке C# для обработки сложных условий допустимо (и широко применяется на практике) вложение операторов `if`. В качестве примера предположим, что необходимо написать программу, определяющую температуру воздуха и возвращающую такую информацию:

- Если температура меньше или равна 32 градусам (по Фаренгейту, то есть 0 °C), программа должна предупреждать водителя, что на дороге будет лед.
- Если температура точно 32 градуса, программа должна сообщать, что на дороге возможны участки, покрытые льдом.
- Если температура больше 32 градусов, программа должна уверить водителя, что льда нет.

Существует много способов написать такую программу. Один из них, с применением вложенных операторов `if`, приведен в примере 3.8.

Пример 3.8. Вложенные операторы if

```
using System;
class Values
{
   static void Main()
```

```
    {
       int temp = 32;

       if (temp <= 32)
       {
          Console.WriteLine("Внимание! Лед на дороге!");
          if (temp == 32)
          {
         Console.WriteLine(
           "Температура замерзания, остерегайтесь воды.");
          }
          else
          {
             Console.WriteLine("Гололед! Температура: {0}", temp);
          }
       }

    }
}
```

Логика этого примера состоит в сравнении температуры со значением 32. Если температура меньше или равна 32, выводится предупреждение:

```
if (temp <= 32)
{
   Console.WriteLine("Внимание! Лед на дороге!");
```

После этого проверяется, равна ли температура 32 градусам. Если это так, выводится одно сообщение, если нет – другое (температура наверняка ниже 32 градусов). Заметим, что второй оператор `if` вложен внутрь первого, так что логика оператора `else` следующая: «поскольку установлено, что температура меньше или равна 32 градусам, и она не равна 32, значит, она меньше 32».

Оператор switch: альтернатива вложенным if

Вложенные операторы `if` трудно читать, трудно правильно написать и трудно отлаживать. При наличии большого списка вариантов выбора оператор `switch` является более мощным альтернативным инструментом. Его логика такова: «выбрать подходящее значение и действовать соответствующим образом».

```
switch (выражение)
{
   case константное-выражение:
      оператор
      оператор-перехода
   [default: оператор]
}
```

Как вы можете видеть, аналогично оператору `if` в заголовке оператора `switch` в круглых скобках приводится выражение. В каждом операто-

Не все операции созданы равными

Более внимательное изучение второго оператора `if` в примере 3.8 позволяет обнаружить потенциальную проблему. Оператор проверяет, равна ли температура 32 градусам:

```
if (temp == 32)
```

В языках C и C++ подобные операторы таят в себе определенную опасность. Неопытные программисты довольно часто пишут операцию присваивания вместо операции сравнения:

```
if (temp = 32)
```

Эту ошибку трудно заметить, а в результате переменной `temp` присваивается значение `32`, и оно же возвращается как значение оператора присваивания. Поскольку в языках C и C++ любое ненулевое значение приравнивается к `true`, оператору `if` всегда возвращается `true`. В качестве побочного эффекта переменная `temp` получит значение `32`, независимо от ее первоначального значения. Это распространенная ошибка, и она привела бы ко множеству проблем, если бы разработчики языка C# не предвидели ее!

В C# проблема решается благодаря требованию, чтобы операторы `if` принимали только логические значения. Число `32`, возвращаемое оператором присваивания, не является логическим (оно целое), а в C# отсутствует автоматическое преобразование `32` в `true`. Ошибка будет обнаружена на этапе компиляции, и это значительный шаг вперед по сравнению с C++, учитывая такую невысокую цену, как отказ от неявного преобразования целых значений в логические!

ре `case` должно присутствовать константное выражение, то есть литеральная или символьная константа либо перечисление.

Если константное выражение соответствует выражению в заголовке, выполняется оператор (или блок операторов), связанный с этим оператором `case`. За выполняемым оператором должен следовать оператор перехода. Как правило, это оператор `break`, который прекращает выполнение оператора `switch`. Альтернативой ему является оператор `goto`, передающий управление другому оператору `case`. Это иллюстрируется в примере 3.9.

Пример 3.9. Оператор switch

```
using System;

class Values
{
   static void Main()
   {
      const int Democrat = 0;
```

```
        const int LiberalRepublican = 1;
        const int Republican = 2;
        const int Libertarian = 3;
        const int NewLeft = 4;
        const int Progressive = 5;

        int myChoice = Libertarian;

        switch (myChoice)
        {
           case Democrat:
              Console.WriteLine("Вы проголосовали за демократов.\n");
              break;
           case LiberalRepublican:  // переход к следующему case
              //Console.WriteLine(
                  //"Либеральные республиканцы голосуют за республиканцев\n");
           case Republican:
              Console.WriteLine("Вы проголосовали за республиканцев.\n");
              break;
           case NewLeft:
              Console.Write("Левые теперь - прогрессивная партия");
              goto case Progressive;
           case Progressive:
             Console.WriteLine("Вы проголосовали за прогрессивную партию.\n");
              break;
           case Libertarian:
              Console.WriteLine("Либертарианцы голосуют за республиканцев");
              goto case Republican;
           default:
              Console.WriteLine("Вы сделали неправильный выбор.\n");
              break;
        }

        Console.WriteLine("Спасибо за участие в выборах.");

     }
  }
```

В этом эксцентричном примере создаются константы для различных политических партий США. Одно из значений (`Libertarian`) присваивается переменной `myChoice`, которая передается оператору `switch`. Когда значение `myChoice` равно `Democrat`, печатается соответствующее сообщение. Обратите внимание, что этот оператор `case` заканчивается оператором `break`. Последний является оператором перехода, который передает управление за пределы оператора `switch`, точнее, на первую строку программы после него. В ней выводится сообщение: «Спасибо за участие в выборах».

Значению `LiberalRepublican` никакие операторы не соответствуют, и выполнение передается на следующий оператор `case`, со значением `Republican`. Передать управление подобным образом можно лишь в том случае, когда у оператора `case` нет никаких выполняемых операторов.

Если снять комментарий с метода `WriteLine` для случая `LiberalRepublican`, код не будет скомпилирован.

Вниманию программистов, пишущих на языке C++! Если оператор `case` не пуст, передать управление на следующий оператор нельзя. Иными словами, такой код корректен:

```
case 1: // переход далее
case 2:
```

В этом примере оператор `case 1` пустой. С другой стороны, код

```
case 1:
    TakeSomeAction();
        // переход далее
case 2:
```

не является корректным, так как оператор `case 1` содержит внутри себя выполняемый оператор. При желании перейти на `case 2` следует сделать это явно, с помощью оператора `goto`:

```
case 1:
   TakeSomeAction();
   goto case 2 // явный переход далее
case 2:
```

В тех случаях, когда требуется выполнить оператор, а затем перейти к другому варианту выбора, необходимо воспользоваться оператором `goto`, как сделано в варианте выбора для `NewLeft`:

```
goto case Progressive;
```

Вовсе не обязательно, чтобы оператор `goto` передавал управление на оператор `case`, непосредственно следующий за текущим. В варианте выбора `Libertarian` оператор `goto` передает управление назад, на оператор `case Republican`. Поскольку оператору `switch` было передано значение `Libertarian`, именно это и происходит при выполнении программы. Выводится сообщение, заготовленное на случай `Libertarian`, выполняется переход на оператор `case Republican`, выводится сообщение, и, наконец, оператор `break` передает управление последнему оператору программы. Окончательный вывод выглядит так:

```
Либертарианцы голосуют за республиканцев
Вы голосовали за республиканцев.

Спасибо за участие в выборах.
```

Обратим внимание на оператор `default` в примере 3.9:

```
default:
    Console.WriteLine(
     "Вы сделали неправильный выбор.\n");
```

Если ни один оператор case не подойдет под значение выражения оператора switch, будет выполнен оператор default. В данном примере он сообщает пользователю об ошибке.

Строковые значения в операторе switch

В рассмотренном примере значение в операторе switch было целым. Язык C# позволяет пользоваться строками:

```
case "Libertarian":
```

Управление передается на оператор case с соответствующей строкой.

Операторы цикла

В языке C# существует широкий выбор операторов цикла. Сюда входят for, while и do...while, а также оператор foreach, новый для семейства C, но известный программистам, пишущим на VB. Кроме того, C# поддерживает операторы перехода goto, break, continue и return.

Оператор goto

Оператор goto – родоначальник всех остальных операторов перехода. К сожалению, он является также источником бесконечной путаницы и «спагетти-кода». Большинство опытных программистов избегают применять его, и он обсуждается здесь лишь ради полноты изложения. Вот как надо им пользоваться:

1. Создайте метку.
2. Укажите эту метку в операторе goto.

Метка – это идентификатор, за которым стоит двоеточие. В типичном случае оператор goto связан с неким условием, как показано в примере 3.10.

Пример 3.10. Оператор goto

```
using System;
public class Tester
 {

     public static int Main()
     {
      int i = 0;
      repeat:            // метка
      Console.WriteLine("i: {0}",i);
      i++;
      if (i < 10)
         goto repeat;  // никогда так не делайте
         return 0;
     }
 }
```

Если попытаться проследить за выполнением программы, насыщенной операторами goto, то она предстанет набором пересекающихся и перекрывающихся участков кода, напоминающих спагетти на тарелке. Отсюда и термин «спагетти-код». Этот феномен привел к появлению альтернативных конструкций, таких как цикл while. Многие программисты чувствуют, что применение goto где-либо, кроме тривиальных учебных примеров, порождает запутанный и плохо сопровождаемый код.

Цикл while

Семантика цикла while такова: «пока это условие истинно, выполнять эту работу».

Синтаксис этого оператора:

```
while (выражение) оператор
```

Как правило, выражение – это какой-нибудь оператор, возвращающий значение. У операторов while выражение обязательно должно возвращать логическое значение (true или false). Естественно, вместо одного оператора в теле while может стоять блок операторов. Пример 3.11 получен в результате переработки примера 3.10 так, что теперь применяется цикл while.

Пример 3.11. Цикл while

```
using System;
public class Tester
 {

     public static int Main()
     {
      int i = 0;
      while (i < 10)
      {
         Console.WriteLine("i: {0}",i);
         i++;
      }
         return 0;
     }
 }
```

Код примера 3.11 выдает тот же результат, что и код примера 3.10, однако его логика яснее. Оператор while является самодостаточным, что весьма удобно. Его смысл: «пока i меньше 10, выводить это сообщение и увеличивать i на 1».

Обратите внимание, что в операторе while значение i проверяется до входа в цикл. Это гарантирует, что цикл не будет выполнен, если условие сразу окажется ложным. Если i изначально равняется 11, цикл выполняться не будет.

Цикл do...while

Возможны ситуации, в которых цикл `while` не отвечает поставленной задаче. Иногда необходимо изменить семантику с «пока условие истинно, работать» на «работать, пока условие остается истинным». Тонкое различие состоит в том, что во втором случае в начале предпринимается действие, а затем, по его окончании, проверяется условие. В таких случаях необходимо применять цикл `do...while`:

```
do оператор while выражение
```

Выражение – любой оператор, возвращающий значение. Пример 3.12 иллюстрирует применение цикла `do...while`.

Пример 3.12. Цикл do...while

```
using System;
public class Tester
{

     public static int Main()
     {
           int i = 11;
           do
           {
               Console.WriteLine("i: {0}",i);
               i++;
          } while (i < 10);
          return 0;
     }
}
```

Здесь `i` инициализируется значением 11, а проверка в операторе `while` дает отрицательный результат только после того, как цикл выполнится один раз.

Цикл for

Внимательное изучение цикла `while` в примере 3.11 позволяет обнаружить некий образец, характерный для итерационных операторов: инициализировать переменную (`i = 0`), проверить ее значение (`i < 10`), выполнить ряд операторов, увеличить переменную (`i++`). Цикл `for` позволяет собрать все эти шаги в одном операторе:

```
for ([инициализация]; [выражение]; [итерация]) оператор
```

Применение цикла `for` показано в примере 3.13.

Пример 3.13. Цикл for

```
using System;
public class Tester
{
```

```
    public static int Main()
    {
        for (int i=0;i<100;i++)
        {
            Console.Write("{0} ", i);

            if (i%10 == 0)
            {
                Console.WriteLine("\t{0}", i);
            }
        }
        return 0;
    }
}
```

Вывод:

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

В коде примера 3.13 использована операция деления по модулю, которая описывается далее в этой главе. Значения переменной `i` выводятся, пока она не станет кратной `10`.

```
if (i%10 == 0)
```

Затем выводится знак табуляции, а после него значение `i`. Таким образом, справа появляются десятки (20, 30, 40 и т. д.).

Отдельные значения выводятся методом `Console.Write`, который во многом похож на `WriteLine`, но не выводит символ новой строки, позволяя выводить значения на той же строке.

Несколько кратких замечаний. В цикле `for` условие проверяется до выполнения операторов. Так, в рассматриваемом примере переменной `i` присваивается ноль, затем проверяется, меньше ли она 100. Поскольку условие `i < 100` истинно, выполняются операторы в теле цикла. После их выполнения переменная `i` инкрементируется (`i++`).

Обратите внимание, что областью видимости переменной `i` является цикл `for` (иными словами, она не видна вне цикла). Код из примера 3.14 компилироваться не будет.

Пример 3.14. Область видимости переменных, определенных в цикле for

```
using System;
public class Tester
```

```
{
     public static int Main()
     {
          for (int i=0; i<100; i++)
          {
               Console.Write("{0} ", i);

               if ( i%10 == 0 )
               {
                    Console.WriteLine("\t{0}", i);
               }
          }
          Console.WriteLine("\n Конечное значение i: {0}", i);
          return 0;
     }
}
```

Строчка, выделенная полужирным шрифтом, вызовет сообщение об ошибке, поскольку переменная `i` недоступна вне цикла `for`.

Символы-разделители и фигурные скобки

Не утихают споры по поводу использования символов-разделителей в программировании. Рассмотрим цикл `for` из примера 3.14:

```
for (int i=0;i<100;i++)
{
   Console.Write("{0} ", i);
   if (i%10 == 0)
   {
      Console.WriteLine("\t{0}", i);
   }
}
```

можно вставить большее количество пробелов:

```
for ( int i = 0; i < 100; i++ )
{
   Console.Write("{0} ", i);
   if ( i % 10 == 0 )
   {
      Console.WriteLine("\t{0}", i);
   }
}
```

Вообще говоря, это дело вкуса. Автор считает, что хотя символы-разделители улучшают читаемость кода, их излишек может иногда сбить с толку. В данной книге код по возможности сжат в целях экономии места на странице.

Оператор foreach

Оператор foreach является новым для семейства языков C. Он применяется для циклического перебора элементов массива или коллекции. Обсуждение этого исключительно полезного оператора откладывается до главы 9.

Операторы continue и break

Бывают ситуации, когда необходимо начать новый проход цикла, не выполнив текущий проход до конца. Оператор continue передает управление в начало тела цикла, после чего выполнение продолжается.

Обратной стороной этой медали является возможность прервать выполнение цикла и немедленно выйти из него. Для этого существует оператор break.

Операторы continue и break приводят к появлению дополнительных точек выхода. Это ухудшает читаемость кода и затрудняет его сопровождение. Не злоупотребляйте этими операторами.

В примере 3.15 иллюстрируется механизм работы операторов continue и break. Этот код, предоставленный автору одним из его рецензентов, Дональдом Кси (Donald Xie), задуман как система обработки сигнала семафора. Сигналы имитируются вводом цифр и букв (в верхнем регистре) с клавиатуры. Для ввода используется метод Console.ReadLine, который считывает текст, введенный с клавиатуры.

Алгоритм довольно прост. Ввод цифры 0 означает нормальные условия – ничего не надо предпринимать, кроме регистрации этого события. (Программа выводит сообщение на консоль, а реальная система могла бы, например, делать запись с отметкой времени в базе данных.) Если принят сигнал Abort (Прекратить), имитируемый буквой «A», этот факт регистрируется и процесс прерывается. Наконец, любое другое событие означает тревогу, возможно, с вызовом полиции. (Этот пример полицию не вызывает, но тревожное сообщение на консоль все-таки выводит.) По сигналу «X» поднимается тревога, и цикл while останавливается.

Пример 3.15. Использование операторов continue и break

```
using System;
public class Tester
{
   public static int Main()
   {
      string signal = "0";       // инициализация: нейтральное состояние
      while (signal != "X")      // X обозначает остановку
      {
         Console.Write("Введите сигнал: ");
```

```
            signal = Console.ReadLine();

            // выполнить какие-либо действия независимо
            // от принятого сигнала
            Console.WriteLine("Принято: {0}", signal);

            if (signal == "A")
            {
               // сбой - прервать обработку сигнала
               // занести проблему в журнал и прерваться
               Console.WriteLine("Сбой! Прекращение работы\n");
               break;
            }

            if (signal == "0")
            {
               // нормальные условия
               // сделать запись в журнале и продолжить работу
               Console.WriteLine("Все хорошо.\n");
               continue;
            }

            // Проблема. Предпринять некоторые действия, сделать
            // запись в журнале и продолжить работу
            Console.WriteLine("{0} -- тревога!\n",
               signal);
         }
         return 0;
      }
   }
```

Вывод:

```
Введите сигнал: 0
Принято: 0
Все хорошо.

Введите сигнал: B
Принято: B
B -- тревога!

Введите сигнал: A
Принято: A
Сбой!. Прекращение работы

Press any key to continue
```

Целью этого примера является демонстрация работы операторов `continue` и `break`. Когда принят сигнал A, выполняются действия в операторе `if`, а затем программа *прерывает* цикл, не поднимая тревогу. Если пришел сигнал 0, тревога тоже не поднимается, а программа *продолжает* выполнение цикла с начала.

Операции

Операция – это некоторое действие, обозначаемое символом. Базовые типы языка C# (например `int`) поддерживают целый ряд операций, таких как присваивание, инкрементирование и др. Их смысл интуитивно понятен, правда, операции присваивания (`=`) и равенства (`==`) легко перепутать.

Операция присваивания (=)

Ранее в этой главе, в разделе «Выражения», было продемонстрировано применение операции присваивания. Символ «=» заставляет операнд, стоящий слева от него, изменить свое значение на значение операнда, стоящего справа от символа «=».

Математические операции

В языке C# существует пять математических операций. Четыре – это стандартные арифметические действия, а пятая возвращает остаток от деления нацело. Эти операции рассматриваются в следующих подразделах.

Простые арифметические операции (+, –, *, /)

C# предлагает четыре простых арифметических операции: сложение (`+`), вычитание (`-`), умножение (`*`) и деление (`/`). Их действие вполне предсказуемо, возможно, за исключением операции деления целых чисел.

При делении двух целых чисел C# поступает, как ученик первого класса, то есть отбрасывает остаток. Тогда результатом деления 17 на 4 будет 4 (`17 / 4 = 4` и `1` в остатке). Для получения остатка в C# предусмотрена специальная операция, деление по модулю (`%`), которая описана в следующем подразделе.

Впрочем, при делении чисел типа `float`, `double` и `decimal` возвращаются дробные ответы.

Операция деления по модулю (%). Получение остатка

Чтобы найти остаток от деления целых чисел, необходимо воспользоваться операцией деления по модулю (`%`). Например, `17 % 4` дает в результате `1` (остаток от деления нацело).

Операция деления по модулю (`%`) оказывается гораздо более полезной, чем можно поначалу ожидать. Если число, кратное n, разделить по модулю на n, то результатом будет 0. Например, `80 % 10 = 0`, потому что 80 делится на 10 без остатка. Это обстоятельство позволяет создавать циклы, в которых действия предпринимаются только на каждом n-ном проходе, для чего достаточно проверять на равенство нулю результат деления по модулю счетчика цикла на число `n`. Подобная стратегия

удобна в цикле `for` и уже применялась ранее в этой главе. Результат операции деления для чисел типов `int`, `float`, `double` и `decimal` показан в примере 3.16.

Пример 3.16. Деление и деление по модулю

```
using System;
class Values
{
   static void Main()
   {
      int i1, i2;
      float f1, f2;
      double d1, d2;
      decimal dec1, dec2;

      i1 = 17;
      i2 = 4;
      f1 = 17f;
      f2 = 4f;
      d1 = 17;
      d2 = 4;
      dec1 = 17;
      dec2 = 4;
      Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n",
          i1/i2, f1/f2);
      Console.WriteLine("double:\t\t{0}\ndecimal:\t{1}",
          d1/d2, dec1/dec2);
      Console.WriteLine("\nModulus:\t{0}", i1%i2);

   }
}
```

Вывод:

```
Integer:        4
float:          4.25
double:         4.25
decimal:        4.25

Modulus:        1
```

Рассмотрим повнимательнее следующую строчку примера 3.16:

```
Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n",
    i1/i2, f1/f2);
```

Она начинается с вызова метода `Console.WriteLine`, которому передается строчка:

```
"Integer:\t{0}\n
```

В результате на консоль выводятся символы `Integer:`, за ними – табуляция (`\t`), далее значение первого параметра (`{0}`) и, наконец, перевод строки (`\n`). Аналогично интерпретируется следующий отрезок параметра:

```
float:\t\t{1}\n
```

Выводится `float:`, два символа табуляции (для выравнивания), значение второго параметра (`{1}`) и символ новой строки. Несколько отличается последний вызов метода:

```
Console.WriteLine("\nModulus:\t{0}", i1%i2);
```

На этот раз параметр начинается с символа новой строки, и перед выводом символов `Modulus:` пропускается пустая строка, что нетрудно наблюдать на экране.

Операции инкрементирования и декрементирования

В программах часто приходится прибавлять к переменной какое-либо значение (или вычитать, или менять переменную как-то иначе) и затем присваивать результат той же переменной. Не исключено, что этот результат заодно необходимо присвоить и другой переменной. В следующих подразделах обсуждается, как поступать в подобных ситуациях.

Операции вычисления и повторного присваивания

Предположим, переменную `mySalary` надо увеличить на 5000. Можно написать:

```
mySalary = mySalary + 5000;
```

Сложение выполняется до присваивания, и нет ничего незаконного в присваивании результата той же переменной. Когда операции будут выполнены, переменная `mySalary` станет больше на 5000. Подобное возможно с любой другой математической операцией:

```
mySalary = mySalary * 5000;
mySalary = mySalary - 5000;
```

и так далее.

Такие действия настолько распространены, что C# предоставляет программисту специальные операции. Сюда входят операции `+=`, `-=`, `*=`, `/=` и `%=`, которые сочетают в себе присваивание и соответственно сложение, вычитание, умножение, деление и деление по модулю. Следовательно, предыдущие примеры можно переписать так:

```
mySalary += 5000;
mySalary *= 5000;
mySalary -= 5000;
```

Их результатом будет соответственно увеличение значения `mySalary` на 5000, умножение той же переменной на 5000 и вычитание из нее 5000.

Поскольку увеличение и уменьшение на 1 встречается довольно часто, в C# (как до того в C и C++) имеются две специальные операции. Уве-

личение на 1 реализуется операцией инкрементирования (++), а уменьшение на 1 – операцией декрементирования (--).

Так, чтобы увеличить значение переменной myAge на 1, можно написать:

```
myAge++;
```

Префиксные и постфиксные операции

Можно выполнить и более сложные действия, например инкрементировать значение переменной и присвоить результат другой переменной:

```
firstValue = secondValue++;
```

Здесь есть одна тонкость. Необходимо ли присвоить значение другой переменной до инкремента или после? Иными словами, пусть переменная secondValue изначально равна 10. Требуется ли, чтобы обе переменные, firstValue и secondValue, были равны 11 или чтобы firstValue получила значение 10, а secondValue – значение 11?

C# (опять-таки, вслед за C и C++) предлагает две версии операций инкрементирования и декрементирования – *префиксную* и *постфиксную*. То есть можно написать:

```
firstValue = secondValue++;  // постфиксная операция
```

и тогда вначале выполняется присваивание, а затем инкрементирование (firstValue=10, secondValue=11). Или же можно написать:

```
firstValue = ++secondValue;  // префиксная операция
```

Здесь вначале происходит инкрементирование, и лишь после этого присваивание (firstValue=11, secondValue=11).

Важно понимать разницу между префиксными и постфиксными операциями, которая иллюстрируется примером 3.17.

Пример 3.17. Префиксное и постфиксное инкрементирование

```
using System;
class Values
{
   static void Main()
   {
      int valueOne = 10;
      int valueTwo;
      valueTwo = valueOne++;
      Console.WriteLine("После постфиксной операции: {0}, {1}", valueOne,
      valueTwo);
      valueOne = 20;
      valueTwo = ++valueOne;
      Console.WriteLine("После префиксной операции: {0}, {1}", valueOne,
      valueTwo);
   }
}
```

Вывод:

```
После постфиксной операции: 11, 10        После префиксной операции: 21, 21
```

Операции отношения

Операции отношения служат для сравнения двух значений и возврата логического значения (`true` или `false`). Например, операция «больше» (`>`) возвращает `true`, если значение слева от символа операции больше значения справа от этого символа. Так, `5 > 2` возвращает значение `true`, а `2 > 5` — `false`.

Операции отношения, принятые в языке C#, перечислены в табл. 3.3. В ней предполагается, что переменной `bigValue` присвоено значение `100`, а переменной `smallValue` — значение `50`.

Таблица 3.3. Операции отношения в языке C# (предполагается, что bigValue = 100, а smallValue = 50)

Название	Операция	Пример	Результат
Равно	`==`	`bigValue == 100` `bigValue = 80`	`true` `false`
Не равно	`!=`	`bigValue != 100` `bigValue != 80`	`false` `true`
Больше	`>`	`bigValue > smallValue`	`true`
Больше или равно	`>=`	`bigValue >= smallValue` `smallValue >= bigValue`	`true` `false`
Меньше	`<`	`bigValue < smallValue`	`false`
Меньше или равно	`<=`	`smallValue <= bigValue` `bigValue <= smallValue`	`true` `false`

Каждая из этих операций действует именно так, как можно ожидать. Впрочем, обратите внимание, что операция «равно» обозначается двумя знаками равенства (`==`) подряд, то есть без пробела между ними. Компилятор C# трактует их как единое целое.

Операция равенства в C# сравнивает объекты, стоящие по обе стороны от знака операции, и возвращает логическое значение `true` или `false`. Таким образом, оператор

```
myX == 5;
```

возвращает `true`, если `myX` является переменной и имеет значение 5.

Многие путают операцию присваивания (`=`) и операцию проверки на равенство (`==`). Последняя обозначается двумя символами «равно», а первая — только одним.

Использование логических операций в условиях

В операторах `if` (рассмотренных ранее в этой главе) проверяется истинность некоторого условия. Нередко возникает необходимость убедиться, что два условия истинны одновременно или что истинно хотя бы одно, или, например, не истинно ни одно из них. На этот случай C# предоставляет целый набор логических операций, приведенных в табл. 3.4. В ней предполагается, что переменная `x` имеет значение `5`, а переменная `y` – значение `7`.

Таблица 3.4. Логические операции языка C# (предполагается, что x = 5, а y = 7)

Название	Операция	Пример	Результат	Логика
И	`&&`	`(x == 3) && (y == 7)`	`false`	Оба операнда должны быть истинны
ИЛИ	`||`	`(x == 3) || (y == 7)`	`true`	Один из двух или оба операнда должны быть истинны
НЕ	`!`	`! (x == 3)`	`true`	Выражение должно быть ложным

Операция И проверяет, чтобы оба ее операнда были истинными. В первой строке табл. 3.4 приведен пример использования операции И:

```
(x == 3) && (y == 7)
```

Выражение в целом имеет значение `false`, поскольку левая часть (`x == 3`) равна `false`.

Что касается операции ИЛИ, либо один ее операнд, либо оба должны быть истинны. Выражение имеет значение `false`, только когда оба операнда равны `false`. Так, в примере из табл. 3.4:

```
(x == 3) || (y == 7)
```

все выражение равно `true`, потому что правая часть (`y == 7`) равна `true`.

Когда выполняется операция НЕ, результат равен `true`, если выражение равно `false`, и наоборот. В соответствующем примере:

```
! (x == 3)
```

выражение равно `true` из-за того, что проверяемое выражение (`x == 3`) равно `false`. (Логика этой операции такова: «истинно утверждение, что не истинно утверждение, что `x` равно 3».)

Приоритет операций

Компилятор должен знать, в каком порядке выполнять операции, когда их несколько. Например, код:

```
myVariable = 5 + 7 * 3;
```

Сокращенное вычисление выражения

Рассмотрим следующий участок кода:

```
int x = 8;
if ((x == 8) || (y == 12))
```

Оператор `if` здесь нетривиален. Проверяемое выражение заключено в круглые скобки, как и положено в языке C#. Оператор `if` будет истинным, если то, что находится внутри внешних скобок, имеет значение `true`.

А внутри этих скобок стоят два выражения (`x == 8`) и (`y == 12`), которые объединены операцией ИЛИ (`||`). Поскольку переменная `x` равна 8, левый операнд равен `true`. Нет необходимости вычислять значение второго операнда (`y == 12`). Все выражение истинно независимо от того, равняется ли `y` двенадцати или нет. Теперь рассмотрим другой участок кода:

```
int x = 8;
if ((x == 5) && (y == 12))
```

И здесь нет необходимости выяснять значение второго операнда. Так как первый операнд ложен, то ложно и все выражение. (Вспомним, что для истинности операции И необходимо, чтобы оба операнда были истинны.)

В случаях, подобных описанным, компилятор C# сокращает вычисление выражения. Вторая проверка не выполняется.

содержит три операции, которые должен выполнить компилятор (`=`, `+` и `*`). В принципе, он мог бы выполнить операции последовательно, то есть присвоить 5 переменной `myVariable`, сложить 5 и 7 и умножить результат на 3. Получится 36, но, конечно, это значение никого не устраивает и будет отброшено. И, естественно, не это подразумевается в приведенном коде.

Правила, устанавливающие приоритет операций, предписывают порядок выполнения этих операций. Как и в алгебре, умножение имеет приоритет над сложением, и 5 + 7 * 3 равно 26, а не 36. Сложение и умножение имеют приоритет над присваиванием, так что компилятор вначале выполнит математические операции и лишь затем присвоит результат (26) переменной `myVariable`.

В C# порядок выполнения операций можно изменить с помощью скобок, подобно тому как это делается в алгебре. Результат рассматриваемого примера можно изменить, написав:

```
myVariable = (5+7) * 3;
```

Группирование элементов оператора присваивания таким образом заставит компилятор сложить 5 и 7, умножить сумму на 3 и присвоить

результат (36) переменной `myVariable`. В табл. 3.5 приводятся приоритеты операций языка C#.

Таблица 3.5. Приоритеты операций

Категория	Операции
Первичные	`(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked stackalloc`
Унарные	`+ - ! ~ ++x --x (T)x *x &x`
Мультипликативные	`* / %`
Аддитивные	`+ -`
Сдвиги	`<< >>`
Операции отношения	`< > <= >= is as`
Равенство	`== !=`
Поразрядное И	`&`
Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ	`^`
Поразрядное ИЛИ	`|`
Логическое И	`&&`
Логическое ИЛИ	`||`
Условная операция (тернарная)	`?:`
Присваивание	`= *= /= %= += -= <<= >>= &= ^= |=`

В некоторых сложных уравнениях приходится вкладывать скобки друг в друга, чтобы гарантировать правильный порядок выполнения операций. Предположим, некий программист захотел узнать, сколько времени его семья тратит впустую по утрам (в секундах).

Оказывается, родители пьют кофе в течение 20 минут и еще 10 минут читают газеты. Дети 30 минут бездельничают и 10 минут спорят друг с другом.

Получается такой алгоритм:

```
(((minDrinkingCoffee  + minReadingNewspaper)* numAdults) +
((minDawdling + minArguing) * numChildren)) * secondsPerMinute
```

Хотя это выражение работает, его трудно читать и понять, что происходит. Код станет намного проще, если воспользоваться промежуточными переменными:

```
wastedByEachAdult = minDrinkingCoffee + minReadingNewspaper;
wastedByAllAdults = wastedByEachAdult * numAdults;
wastedByEachKid = minDawdling  + minArguing;
wastedByAllKids = wastedByEachKid * numChildren;
wastedByFamily = wastedByAllAdults + wastedByAllKids;
totalSeconds = wastedByFamily * 60;
```

Здесь довольно много промежуточных переменных, но код легче читать, понимать и, самое важное, отлаживать. Если выполнить эту программу в отладчике, можно будет просматривать значения промежуточных переменных и убеждаться в их правильности.

Тернарная операция

В то время как у большинства операций один или два операнда (например, `myValue++` или `a + b` соответственно), существует одна операция, требующая три операнда. Она называется тернарной, обозначается символами `?:` и имеет следующий синтаксис:

условное-выражение **?** *выражение1* **:** *выражение2*

В этой операции анализируется *условное-выражение* (выражение, возвращающее значение типа `bool`) и выполняется одно из двух других выражений. Если условное выражение возвратило `true`, выполняется *выражение1*, если *false* – *выражение2*. Семантика этой операции: «если условие истинно, выполнить первое действие; в противном случае – второе». Иллюстрация приводится в примере 3.18.

Пример 3.18. Тернарная операция

```
using System;
class Values
{
   static void Main()
   {
      int valueOne = 10;
      int valueTwo = 20;

        int maxValue = valueOne > valueTwo ?  valueOne : valueTwo;

        Console.WriteLine("ValueOne: {0}, valueTwo: {1}, maxValue: {2}",
            valueOne, valueTwo, maxValue);

   }
}
```

Вывод:

```
ValueOne: 10, valueTwo: 20, maxValue: 20
```

В примере 3.18 тернарная операция используется для проверки, какое значение больше, `valueOne` или `valueTwo`. Если первое, то оно присваивается переменной `maxValue`. В противном случае `maxValue` получает значение `valueTwo`.

Пространства имен

В главе 2 приводились причины появления в языке C# такого понятия, как пространства имен (например, желание избежать конфликта имен при обращении к библиотекам разных производителей). В дополнение к пространствам имен, предоставляемым платформой .NET Fra-

mework и другими производителями программного продукта, программист может создавать собственные. Это делается с помощью ключевого слова namespace, за которым следует имя создаваемого пространства. Объекты этого пространства необходимо заключать в фигурные скобки, как показано в примере 3.19.

Пример 3.19. Создание пространства имен

```
namespace Programming_C_Sharp
{
   using System;
   public class Tester
   {

      public static int Main()
      {
         for (int i=0;i<10;i++)
         {
            Console.WriteLine("i: {0}",i);
         }
         return 0;
      }
   }
}
```

В примере 3.19 создается пространство имен, названное Programming_C_Sharp, и объявляется класс Tester, существующий в этом пространстве. В языке C# допустимо вложение пространств имен, то есть объявление одного пространства имен внутри другого. Это может понадобиться для сегментирования кода. Объекты, созданные в пределах внутреннего пространства, будут защищены от обращения к ним из внешнего пространства. Это демонстрирует пример 3.20.

Пример 3.20. Вложение пространств имен

```
namespace Programming_C_Sharp
{
   namespace Programming_C_Sharp_Test
   {
      using System;
      public class Tester
      {

           public static int Main()
           {
               for (int i=0;i<10;i++)
               {
                  Console.WriteLine("i: {0}",i);
               }
              return 0;
           }
       }
   }
}
```

Теперь объект Tester объявлен в пространстве имен Programming_C_Sharp_Test:

```
Programming_C_Sharp.Programming_C_Sharp_Test.Tester
```

Это имя не будет конфликтовать ни с каким другим объектом Tester в других пространствах имен, в том числе в пространстве Programming_C_Sharp.

Директивы препроцессора

Примеры, приводимые ранее в этой книге, компилировались целиком, если вообще компилировались. Однако бывают ситуации, когда требуется скомпилировать только часть программы в зависимости, например, от того, отлаживается ли она или уже готова к распространению.

До компиляции кода специальная программа, называемая препроцессором, подготавливает его для компилятора. Препроцессор ищет в коде специальные директивы, начинающиеся с символа «#». Эти директивы позволяют программисту определять идентификаторы и проверять их существование.

Определение идентификаторов

Конструкция #define DEBUG определяет идентификатор препроцессора с именем DEBUG. В то время как прочие директивы препроцессора могут находиться в любом месте кода, идентификаторы должны быть определены в самом начале, даже до операторов using.

Директива #if позволяет проверить, определен ли идентификатор DEBUG. Например, можно написать следующий код:

```
#define DEBUG

//... обычный код, не затрагиваемый препроцессором

#if DEBUG
   // код, компилируемый при отладке
#else
   // код, компилируемый, если отладка не происходит
#endif

//... обычный код, не затрагиваемый препроцессором
```

Когда работает препроцессор, он встречает оператор #define и запоминает идентификатор DEBUG. Препроцессор пропускает обычный код C# и находит блок #if – #else – #endif.

В директиве #if проверяется существование идентификатора DEBUG. Поскольку он определен, код между #if и #else компилируется, а код между #else и #endif *не* компилируется. Он вообще не появляется в сборке, словно его не было в исходном тексте.

Если бы директива #if дала отрицательный результат, то есть идентификатор не был определен, то код между #if и #else не был бы скомпилирован. Зато C# скомпилировал бы код между #else и #endif.

Любой код вне конструкции #if – #endif игнорируется препроцессором и компилируется безусловно.

Отмена определения идентификатора

Отмена определения идентификатора выполняется директивой препроцессора #undef. Препроцессор обрабатывает код сверху вниз, так что идентификатор считается определенным от директивы #define до директивы #undef или до конца программы. Таким образом, если написать код:

```
#define DEBUG

#if DEBUG
   // этот код будет скомпилирован
#endif

#undef DEBUG

#if DEBUG
   // этот код не будет скомпилирован
#endif
```

то первая директива #if выдаст положительный результат проверки (идентификатор DEBUG определен), а вторая – отрицательный (определение DEBUG отменено).

Директивы #if, #elif, #else и #endif

Оператора, аналогичного switch, у препроцессора нет, но директивы #elif и #else обеспечивают большую гибкость препроцессорной обработки кода. Директива #elif реализует логику «else-if», то есть «если DEBUG существует, предпринять одно действие; иначе, если TEST существует, предпринять другое; иначе – третье»:

```
#if DEBUG
   // скомпилировать этот код, если DEBUG определен
#elif TEST
   // скомпилировать этот код, если DEBUG не определен,
   // но TEST определен
#else
   // скомпилировать этот код, если ни DEBUG
   // ни TEST не определены
#endif
```

В этом примере препроцессор сперва проверяет, определен ли идентификатор DEBUG. Если да, код между #if и #elif будет скомпилирован, а код между #elif и #endif компилироваться не будет.

В том (и только в том) случае, когда DEBUG не определен, препроцессор проверяет TEST. Обратите внимание, что TEST не проверяется, если DEBUG определен. Если идентификатор TEST определен, компилируется код от #elif до #else. Когда не определены ни DEBUG ни TEST, компилируется код, расположенный между директивами #else и #endif.

Директива #region

Директива препроцессора #region помечает область текста комментарием. Основное предназначение этой директивы – позволить инструментальным приложениям, таким как Visual Studio .NET, размечать участки кода и сворачивать их в редакторе, чтобы был виден только комментарий.

Например, при создании Windows-приложения (обсуждается в главе 13) среда Visual Studio .NET создает область для кода, написанного разработчиком. Когда эта область развернута, она выглядит так, как изображено на рис. 3.1. (Замечание: прямоугольник с закругленными углами добавлен автором, чтобы читателю было легче обнаружить область.)

```
/// </summary>
public override void Dispose()
{
   base.Dispose();
   if(components != null)
      components.Dispose();
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
   this.components = new System.ComponentModel.Container();
   this.Size = new System.Drawing.Size(300,300);
   this.Text = "Form1";
}
#endregion

/// <summary>
/// The main entry point for the application.
/// </summary>
```

Рис. 3.1. *Область кода развернута в окне Visual Studio .NET*

На рисунке видна область, помеченная директивами препроцессора `#region` и `#endregion`. Однако когда область свернута, виден лишь комментарий, которым она отмечена (`Windows Form Designer generated code`). Это показано на рис. 3.2.

```
HelloWindows.Form1                    InitializeComponent()

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        public override void Dispose()
        {
            base.Dispose();
            if(components != null)
                components.Dispose();
        }

        Windows Form Designer generated code

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

Рис. 3.2. *Область кода свернута*

4

Классы и объекты

В главе 3 обсуждалось множество базовых типов языка C# (`int`, `long`, `char` и др.). Все же сердцем и душой языка C# является возможность создавать новые, сложные, программно-определяемые типы, четко описывающие объекты, используемые для решения поставленной задачи.

Именно эта возможность создавать новые типы характеризует объектно-ориентированный язык. Программист создает новые типы C#, объявляя и определяя классы. Как будет показано в главе 8, типы можно определять и через интерфейсы. Экземпляры класса называются *объектами*. Объекты создаются в памяти компьютера при выполнении программы.

Разница между классом и объектом такая же, как между понятием «собака» и конкретной собакой, сидящей у ног читателя. Вы не можете играть с описанием собаки, а вот с конкретной собакой это вполне возможно.

Класс `Dog` описывает характеристики собак – вес, рост, цвет глаз, окрас шерсти, их характеры и т. д. Они также могут выполнять некоторые действия: есть, ходить, лаять и спать. Конкретный пес (возьмем в качестве примера Майло, собаку автора) имеет конкретный вес (62 фунта), рост (22 дюйма), цвет глаз (черный), окрас (светлый), характер (ангельский) и т. д. Он может делать все, что умеет делать любая собака (впрочем, при ближайшем знакомстве с ним создается впечатление, что он реализует только метод «есть»).

Неоценимым достоинством классов в объектно-ориентированном программировании является то, что они инкапсулируют характеристики и возможности какой-либо сущности в одной, автономной и самодостаточной *программной единице*. Если понадобится отсортировать со-

держимое экземпляра элемента управления Windows «список», нужно будет лишь выдать ему указание отсортировать самого себя. Неважно, как именно он это сделает; главное – результат. Инкапсуляция, полиморфизм и наследование – три главных принципа объектно-ориентированного программирования.

Есть старая программистская шутка. Сколько объектно-ориентированных программистов нужно для замены перегоревшей лампочки? Ответ: ни одного; достаточно дать указание лампочке заменить себя. (Другой ответ: ни одного; фирма Microsoft изменила стандарт, и теперь должно быть темно.)

В этой главе описываются функциональные возможности языка C#, которые используются для определения новых классов. Свойства класса и действия, которые он выполняет (его поведение), в совокупности называются *членами класса* (*class members*). В данной главе показано, как с помощью методов определять поведение класса и как текущее состояние класса отражается в переменных класса (часто называемых *полями*). Кроме того, в данной главе вводится такое понятие, как *свойства* (*properties*). С точки зрения создателя класса они аналогичны методам, но для пользователей класса они выглядят как поля.

Определение классов

Чтобы определить новый тип или класс, следует вначале объявить его, а затем определить его методы и поля. Класс объявляется с помощью ключевого слова `class`. Полный синтаксис такой:

```
[атрибуты] [модификаторы-прав-доступа] class идентификатор [:базовый-класс]
{тело-класса}
```

Атрибуты обсуждаются в главе 18; модификаторы прав доступа – в следующем разделе. (В большинстве случаев в качестве модификатора прав доступа указывается ключевое слово `public`.) *Идентификатор* – это имя, которое программист дает классу. Необязательная конструкция *базовый-класс* обсуждается в главе 5. Определения элементов, образующие *тело-класса*, заключены в фигурные скобки ({}).

Вниманию программистов, пишущих на языке C++! Определение класса в языке C# *не надо* заканчивать точкой с запятой. Тем не менее, если ее поставить, программа будет компилироваться.

В языке C# все происходит в рамках какого-нибудь класса. Так, в некоторых примерах из главы 3 упоминается класс по имени `Tester`:

```
public class Tester
{
```

```
        public static int Main()
        {
         /...
        }
}
```

До сих пор экземпляры этого класса не создавались, то есть не создавались объекты `Tester`. Какая разница между классом и экземпляром класса? Чтобы ответить на этот вопрос, обсудим разницу между *типом* `int` и *переменной* типа `int`. В то время как можно написать:

```
int myInteger = 5;
```

нельзя написать:

```
int = 5;
```

Нельзя присвоить значение типу, зато можно присвоить значение объекту этого типа (в данном случае – переменной типа `int`).

Объявляя новый класс, программист определяет свойства всех объектов этого класса и их поведение. Пусть, например, программист разрабатывает оконную среду. Для упрощения взаимодействия пользователя с приложением он создает экранные элементы, известные в Windows как элементы управления. Одним из таких элементов является список, предоставляющий пользователю возможность выбора одной из альтернатив.

Список имеет целый ряд характеристик, например высоту, ширину, положение на экране и цвет текста. Программист вправе ожидать, что списки придерживаются определенной линии поведения: их можно открыть, закрыть, отсортировать и т. д.

Объектно-ориентированное программирование позволяет создать новый тип – назовем его `ListBox`, – который инкапсулирует характеристики и поведение окон списков. Такой класс может иметь переменные с именами `height`, `width`, `location` и `text_color` (высота, ширина, место на экране и цвет текста соответственно), а также методы `sort()`, `add()`, `remove()` и т. д.

Типу `ListBox` нельзя присваивать данные. Необходимо создать объект этого типа, как показано в следующем фрагменте кода:

```
ListBox myListBox;
```

Создав экземпляр типа `ListBox`, программист может присваивать данные его полям.

Рассмотрим класс, в котором на экран выводится текущее время. Внутреннее состояние этого класса должно позволять представлять год, месяц, день месяца, часы, минуты и секунды. Желательно, чтобы класс был способен выводить время в разных форматах. Такой класс можно реализовать, определив один метод и шесть переменных, как показано в примере 4.1.

Пример 4.1. Простой класс Time

```
using System;

public class Time
{
   // открытые методы
   public void DisplayCurrentTime()
   {
      Console.WriteLine(
         "Заглушка для DisplayCurrentTime");
   }

    // закрытые переменные
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;

}

public class Tester
{
   static void Main()
   {
      Time t = new Time();
      t.DisplayCurrentTime();
   }

}
```

Единственный метод, объявленный в определении класса `Time`, – метод `DisplayCurrentTime()`. Тело этого метода определено внутри определения класса. В отличие от других языков (например, C++), C# не требует объявления методов до их описания. Кроме того, язык не поддерживает размещение объявлений в одном файле, а кода – в другом (в C# отсутствуют файлы заголовков). В C# все методы встраиваются в определение класса, как метод `DisplayCurrentTime()` в примере 4.1.

В определении метода `DisplayCurrentTime()` указано, что тип возвращаемого значения – `void`, то есть он не возвращает значения методу, вызвавшему его. В примере 4.1 тело метода пока что представляет собой заглушку.

В конце определения класса `Time` объявляется ряд переменных: `Year`, `Month`, `Date`, `Hour`, `Minute` и `Second`.

После закрывающей фигурной скобки определен другой класс, `Tester`. Он включает в себя знакомый читателю метод `Main()`. В методе `Main()` создается экземпляр класса `Time`, и его адрес присваивается объекту `t`. Поскольку `t` является экземпляром `Time`, метод `Main()` может вызывать

метод DisplayCurrentTime(), предоставляемый объектами этого типа, и тем самым выводить на экран текущее время:

```
t.DisplayCurrentTime();
```

Модификаторы прав доступа

Модификатор прав доступа определяет, каким методам класса (а также методам других классов) доступны переменные и методы, определенные в классе. Модификаторы прав доступа перечислены в табл. 4.1.

Таблица 4.1. Модификаторы права доступа

Модификатор права доступа	Ограничения
public	Никаких ограничений. Элементы, определенные как public (открытые), доступны любому методу любого класса
private	Элементы класса A, определенные как private (закрытые), доступны только методам класса A
protected	Элементы класса A, определенные как protected (защищенные), доступны методам класса A и методам классов, *производных от* класса A
internal	Элементы класса A, определенные как internal (внутренние), доступны методам любого класса в сборке класса A
protected internal	Элементы класса A, определенные как protected internal (защищенные внутренние), доступны методам класса A, методам классов, *производных от* класса A, а также методам любого класса в сборке класса A. Этот модификатор имеет семантику «защищенные ИЛИ внутренние». (Понятие «защищенные И внутренние» отсутствует)

Вообще говоря, желательно определять переменные класса как закрытые (private). Это гарантирует, что только методы того же класса будут иметь доступ к их значениям. Поскольку уровень доступа private устанавливается по умолчанию, нет необходимости указывать его явно. Тем не менее автор рекомендует делать это. Так, в примере 4.1 объявления переменных лучше переписать следующим образом:

```
// закрытые переменные
private int Year;
private int Month;
private int Date;
private int Hour;
private int Minute;
private int Second;
```

Класс Time и метод DisplayCurrentTime() объявлены открытыми, чтобы любой другой класс мог обратиться к ним.

Явное указание уровня доступа всех методов и элементов класса характеризует хороший стиль программирования. Хотя программист может положиться на устанавливаемый по умолчанию модификатор доступа `private`, явное указание модификатора права доступа свидетельствует о сознательном выборе и способствует самодокументированию программы.

Аргументы метода

Методы могут принимать любое количество параметров.[1] Список параметров указывается в круглых скобках после имени метода, причем каждый параметр предваряется своим типом. Например, в следующем участке кода определяется метод по имени `MyMethod`, который возвращает `void` (то есть не возвращает значения). Он имеет два параметра с типами `int` и `button`:

```
void MyMethod (int firstParam, button secondParam)
{
  // ...
}
```

В теле метода параметры ведут себя как локальные переменные, как будто бы они были объявлены и инициализированы значениями, переданными методу. В примере 4.2 иллюстрируется передача значений методу. В этом случае значения имеют типы `int` и `float`.

Пример 4.2. Передача значений методу SomeMethod( )

```
using System;

public class MyClass
{
   public void SomeMethod(int firstParam, float secondParam)
   {
      Console.WriteLine(
         "Получены параметры: {0}, {1}",
         firstParam, secondParam);
   }

}
public class Tester
{
```

[1] Термины «аргумент» и «параметр» обычно обозначают одно и то же понятие. Тем не менее, некоторые программисты настаивают на проведении различия между аргументами в объявлении метода и параметрами, передаваемыми ему при вызове. (В русском языке все определено с точностью до наоборот: аргументы передаются при вызове, а параметры – при определении. – *Примеч. науч. ред.*)

```
    static void Main()
    {
        int howManyPeople = 5;
        float pi = 3.14f;
        MyClass mc = new MyClass();
        mc.SomeMethod(howManyPeople, pi);
    }

}
```

Здесь метод SomeMethod() принимает значения типов int и float и выводит их с помощью метода Console.WriteLine(). Параметры, названные firstParam и secondParam, в теле метода SomeMethod() трактуются как локальные переменные.

В вызывающем методе Main() создаются и инициализируются две локальные переменные (howManyPeople и pi). Они передаются методу SomeMethod() в качестве параметров. Компилятор отображает howManyPeople в firstParam, а pi – в secondParam, на основании их позиций в списке параметров.

Создание объектов

В главе 3 было проведено различие между размерными и ссылочными типами. Базовые типы C# (int, char и т. д.) являются размерными типами; они хранятся в стеке. Объекты, со своей стороны, имеют ссылочный тип и хранятся в куче, а создаются с помощью ключевого слова new:

```
Time t = new Time();
```

Здесь переменная t, на самом деле, не содержит значение объекта Time. Вместо этого она содержит адрес объекта (безымянного), созданного в куче. Сама же переменная t является лишь ссылкой на этот объект.

Конструкторы

Обратите внимание на оператор, создающий объект Time в примере 4.1. Внешне он выглядит как вызов метода:

```
Time t = new Time();
```

И действительно, при создании экземпляра происходит обращение к методу. Он называется *конструктором*, и программист должен либо определить его как часть определения класса, либо положиться в этом на среду CLR, которая сама его предоставит. Задача конструктора – создать объект указанного класса и перевести его в *действующее* состояние. До вызова конструктора объект представляет собой неинициализированную область памяти; по окончании его работы эта память содержит действующий экземпляр данного класса.

Класс Time из примера 4.1 не определяет никакого конструктора. Если конструктор не объявлен, компилятор предоставляет его самостоятельно. Конструктор, вызванный по умолчанию, создает объект, но не предпринимает никаких других действий. Переменные класса инициализируются безвредными значениями (целые – нулем, строки – пустой строкой и т. д.). Значения, присваиваемые базовым типам по умолчанию, перечислены в табл. 4.2.

Таблица 4.2. Базовые типы и их значения по умолчанию

Типы	Значение по умолчанию
Числовые (int, long и т. д.)	0
bool	false
char	'\0' (null)
enum	0
Ссылочные	null

Как правило, программисты предпочитают определять собственные конструкторы и снабжать их аргументами, чтобы конструктор мог устанавливать начальное состояние объекта. В примере 4.1 было бы разумно передавать конструктору информацию о текущем годе, месяце, дне месяца и т. д., чтобы объект был заполнен осмысленными данными.

При определении конструктора объявляется метод с тем же именем, что и класс, в котором он определяется. У конструкторов нет возвращаемого типа, и они обычно объявляются открытыми. Если планируется передавать конструктору какие-либо аргументы, их список указывается, как для любого другого метода. В примере 4.3 объявляется конструктор для класса Time, который принимает единственный аргумент, объект типа DateTime.

Пример 4.3. Объявление конструктора

```
public class Time
{
   // открытые методы доступа
   public void DisplayCurrentTime()
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }

   // конструктор
   public Time(System.DateTime dt)
   {

      Year = dt.Year;
      Month = dt.Month;
      Date = dt.Day;
```

```
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;
    }

    // закрытые переменные класса
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;

}

public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time(currentTime);
        t.DisplayCurrentTime();
    }

}
```

Вывод:

```
11/16/2005 16:21:40
```

В этом примере конструктор принимает объект DateTime и инициализирует все переменные класса, беря за основу значения из этого объекта. После того как конструктор закончит свою работу, в памяти компьютера будет находиться объект Time с инициализированными значениями. Когда метод Main() вызывает метод DisplayCurrentTime(), эти значения выводятся на экран.

Закомментируйте какое-нибудь присваивание и снова выполните программу. Окажется, что компилятор инициализирует соответствующую переменную нулем. Целые переменные класса устанавливаются в 0, если не указано другое значение. Помните, что переменные, имеющие размерный тип (например целочисленные), не могут оставаться *неинициализированными*; если конструктор не получит конкретных указаний, он установит значения по умолчанию.

В примере 4.3 в методе Main() класса Tester создается объект DateTime. Этот объект, поставляемый библиотекой System, предлагает ряд открытых значений: Year, Month, Day, Hour, Minute и Second, которые в точности соответствуют переменным объекта Time. Кроме того, объект DateTime предоставляет статический метод Now(), который возвращает ссылку на экземпляр объекта DateTime, инициализированный текущим временем.

Рассмотрим выделенную строчку в методе Main(), где объект DateTime создается вызовом статического метода Now(). Он создает DateTime в куче и возвращает ссылку на него.

Возвращенная ссылка присваивается переменной currentTime, которая объявлена как ссылка на объект DateTime. Затем currentTime передается в качестве параметра конструктору Time. Параметр этого конструктора, dt, тоже представляет собой ссылку на объект DateTime. Теперь dt ссылается на тот же объект, что и переменная currentTime. Таким образом, конструктор Time получает доступ к открытым переменным объекта DateTime, созданного в методе Tester.Main().

Нетрудно объяснить, почему объект DateTime, на который ссылается код конструктора Time, является тем же самым объектом, на который ссылается код метода Main(). Дело в том, что объекты являются экземплярами *ссылочных типов*. Таким образом, когда объект передается в качестве параметра, он передается *по ссылке*, то есть передается указатель на объект, а копия объекта не создается.

Инициализаторы

Вместо того чтобы инициализировать значения переменных класса в каждом конструкторе, можно сделать это в *инициализаторе*. Инициализатор создается присваиванием начального значения элементу класса:

```
private int Second  = 30;  // инициализатор
```

Предположим, семантика объекта Time такова, что независимо от установленного времени секунды всегда инициализируются значением 30. Класс Time можно переписать с применением инициализатора. Тогда, какой бы конструктор ни был вызван, переменная Second будет всегда получать начальное значение либо явно, от конструктора, либо неявно, от инициализатора. Это продемонстрировано в примере 4.4.

Пример 4.4. Использование инициализатора

```
  public class Time
{
   // открытые методы
   public void DisplayCurrentTime()
   {
      System.DateTime now = System.DateTime.Now;
         System.Console.WriteLine(
         "\nОтладка\t: {0}/{1}/{2} {3}:{4}:{5}",
         now.Month, now.Day,  now.Year, now.Hour,
             now.Minute, now.Second);

      System.Console.WriteLine("Время\t: {0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }
```

```
   // конструкторы
   public Time(System.DateTime dt)
   {

      Year =      dt.Year;
      Month =     dt.Month;
      Date =      dt.Day;
      Hour =      dt.Hour;
      Minute =    dt.Minute;
      Second =    dt.Second;   // явное присваивание
   }

     public Time(int Year, int Month, int Date,
         int Hour, int Minute)
     {
         this.Year =     Year;
         this.Month =    Month;
         this.Date =     Date;
         this.Hour =     Hour;
         this.Minute =   Minute;
     }

    // закрытые переменные класса
   private int Year;
   private int Month;
   private int Date;
   private int Hour;
   private int Minute;
   private int Second  = 30;  // инициализатор
}

public class Tester
{
   static void Main()
   {
      System.DateTime currentTime = System.DateTime.Now;
      Time t = new Time(currentTime);
      t.DisplayCurrentTime();

      Time t2 = new Time(2005,11,18,11,45);
      t2.DisplayCurrentTime();

   }
}
```

Вывод:

```
Отладка   : 11/27/2005 7:52:54
Время    : 11/27/2005 7:52:54

Отладка   : 11/27/2005 7:52:54
Время    : 11/18/2005 11:45:30
```

Если инициализатор не указан, конструктор присвоит каждой переменной нулевое начальное значение. Однако в приведенном примере переменная Second получает значение 30:

```
private int Second  = 30;  // инициализатор
```

Если для переменной Second не будет передано значение, то при создании t2 она будет проинициализирована числом 30:

```
Time t2 = new Time(2005,11,18,11,45);
t2.DisplayCurrentTime();
```

Однако если переменной Second значение присвоено, как это делается в конструкторе, принимающем объект DateTime (строчка выделена полужирным шрифтом), новое значение замещает первоначальное.

При первом выполнении программы вызывается конструктор, принимающий объект DateTime, причем секунды устанавливаются в значение 54. В следующий раз явно указывается время 11:45 (секунды опущены), и инициализатор делает свое дело.

Если бы у программы не было инициализатора и переменной Second не присваивалось никакого значения, то компилятор установил бы ее в ноль.

Копирующие конструкторы

Копирующий конструктор (*copy constructor*) создает новый объект, копируя переменные из существующего объекта того же типа. Пусть, например, требуется передать объект Time конструктору Time() так, чтобы новый объект Time содержал те же значения, что и старый.

Язык C# не добавляет в класс копирующий конструктор, так что программист должен написать такой конструктор самостоятельно. Подобный конструктор всего лишь копирует элементы исходного объекта во вновь создаваемый:

```
public Time(Time existingTimeObject)
{
    Year = existingTimeObject.Year;
    Month = existingTimeObject.Month;
    Date = existingTimeObject.Date;
    Hour = existingTimeObject.Hour;
    Minute = existingTimeObject.Minute;
    Second = existingTimeObject.Second;
}
```

Копирующий конструктор вызывается путем создания объекта типа Time и передачи ему имени копируемого объекта Time:

```
Time t3 = new Time(t2);
```

Здесь переменная t2 передается в качестве аргумента existingTimeObject копирующему конструктору, который создаст новый объект Time (переменная t3).

Ключевое слово this

Ключевое слово this является ссылкой на текущий экземпляр объекта. Ссылка this (иногда ее называют указателем[1]) является скрытым указателем на каждый нестатический метод класса. Любой метод может использовать ключевое слово this для доступа к другим нестатическим методам и переменным этого объекта.

Существует три типичных случая применения ссылки this. Первый – доступ к члену объекта, скрытому параметром, как в следующем примере:

```
public void SomeMethod(int hour)
{
   this.hour = hour;
}
```

Здесь метод SomeMethod() принимает параметр hour с тем же именем, что и у переменной класса. Ссылка this используется для устранения двусмысленности. В то время как this.hour обозначает переменную класса, hour обозначает параметр.

В пользу такого стиля говорит удачное имя как для параметра, так и для переменной класса. Контраргумент сводится к тому, что некоторая путаница все же имеет место.

Вторым применением ссылки this является передача ссылки на текущий объект другому методу в качестве аргумента. Например, в следующем коде:

```
public void FirstMethod(OtherClass otherObject)
{
   otherObject.SecondMethod(this);
}
```

используются два класса: один с методом FirstMethod(), а другой, класс OtherClass, с методом SecondMethod(). В теле FirstMethod() вызывается метод SecondMethod(), которому передается текущий объект для дальнейшей обработки.

Третье применение ссылки this связано с индексаторами, обсуждаемыми в главе 9.

---

[1] Указатель – это переменная, в которой хранится адрес объекта в памяти. C# не использует указатели на управляемые объекты (объекты, определенные в управляемом коде).

Статические члены класса

Свойства и методы класса могут быть либо членами *экземпляра* (*instance members*), либо *статическими членами* (*static members*). Первые связаны с экземплярами класса, а вторые рассматриваются как часть самого класса. Программист обращается к статическому члену, указывая имя класса, в котором этот член был объявлен. Пусть, например, имеется класс с именем `Button`. Объекты этого класса названы `btnUpdate` и `btnDelete`. Предположим далее, что класс `Button` имеет статический метод `SomeMethod()`. Чтобы обратиться к статическому методу, следует написать:

```
Button.SomeMethod();
```

а не

```
btnUpdate.SomeMethod();
```

В C# недопустимо обращаться к статическому методу или переменной через экземпляр, и подобные попытки будут пресечены компилятором (программисты, привыкшие к C++, возьмите на заметку!).

В некоторых языках проводится различие между методами класса и другими (глобальными) методами, доступными вне контекста любого класса. В C# нет глобальных методов; существуют только методы класса. Однако аналогичный результат достигается путем определения статических методов в рамках класса.

Статические методы подобны глобальным в том смысле, что программист может вызвать их, не имея под рукой конкретный объект. Однако статические методы имеют перед глобальными то преимущество, что область видимости их имени ограничена классом, и глобальное пространство имен не засоряется многочисленными функциями. Это позволяет сопровождать весьма сложные программы, причем имя класса выполняет задачи пространства имен для статических методов.

Не поддавайтесь искушению создать в программе один класс и «свалить» в него все методы. Это возможно, но крайне нежелательно, поскольку подорвет принципы инкапсуляции объектно-ориентированной разработки.

Вызов статических методов

Метод `Main()` является статическим. Про статические методы говорят, что они действуют в классе, а не в экземпляре класса. Они не имеют ссылки `this`, поскольку отсутствует экземпляр, на который она могла бы указывать.

Статические методы не имеют непосредственного доступа к нестатическим членам. Чтобы метод `Main()` мог вызвать нестатический метод,

он должен создать объект. Вспомним пример 4.2, воспроизведенный здесь для удобства читателя:

```
using System;

public class MyClass
{
   public void SomeMethod(int firstParam, float secondParam)
   {
      Console.WriteLine(
         "Получены параметры: {0}, {1}",
         firstParam, secondParam);
   }

}

public class Tester
{
   static void Main()
   {
      int howManyPeople = 5;
      float pi = 3.14f;
      MyClass mc = new MyClass();
      mc.SomeMethod(howManyPeople, pi);
   }

}
```

`SomeMethod()` представляет собой нестатический метод класса `MyClass`. Чтобы обратиться к этому методу, метод `Main()` должен сначала создать объект типа `MyClass`, а затем вызвать метод данного объекта.

Применение статических конструкторов

Если в классе объявлен статический конструктор, можно гарантировать, что этот конструктор сработает до создания какого бы то ни было экземпляра этого класса.

Программист не в состоянии управлять моментом выполнения статического конструктора, однако можно быть уверенным, что такой конструктор будет выполнен после запуска программы, но до создания первого экземпляра класса. По этой причине нельзя предполагать (и выяснить), создается ли экземпляр в данный момент или нет.

Например, в класс `Time` можно добавить следующий конструктор:

```
static Time()
{
     Name = "Время";
}
```

Обратите внимание на отсутствие модификатора права доступа (например, public) перед статическим конструктором. Модификаторы права доступа в этом случае не разрешены. Кроме того, поскольку данный метод статический, ему недоступны нестатические переменные класса, и переменная Name должна быть объявлена как статическая переменная класса.

```
private static string Name;
```

Заключительным штрихом будет добавление строчки в метод DisplayCurrentTime():

```
public void DisplayCurrentTime()
{
   System.Console.WriteLine("Имя: {0}", Name);
   System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
      Month, Date, Year, Hour, Minute, Second);
}
```

В результате всех этих изменений получается следующий вывод:

```
Имя: Время
11/27/2005 7:52:54
Имя: Время
11/18/2005 11:45:30
```

(Вывод на вашей машине может быть иным, в зависимости от даты и времени запуска этого кода.)

Программа работает, но для достижения той же цели вовсе не обязательно создавать статический конструктор. Вместо этого можно было бы применить инициализатор:

```
private static string Name = "Время";
```

делающий практически то же самое. Статические конструкторы подходят в основном для настройки, которую нельзя выполнить с помощью инициализатора и которая производится только один раз.

Пусть, например, в приложении используется неуправляемый объект из динамической библиотеки (dll), для которого требуется создать класс оболочки. Программист может вызвать загружаемую библиотеку из своего статического конструктора и инициализировать в нем таблицу переходов. Обработка наследуемого кода и взаимодействие с неуправляемым кодом обсуждаются в главе 22.

Применение закрытых конструкторов

В языке C# нет глобальных методов и констант. Некоторые программисты пишут маленькие вспомогательные классы, единственным предназначением которых является хранение статических элементов. Оставим в стороне вопрос о том, насколько это удачное решение. Если

уж такой класс написан, программист определенно не хочет, чтобы создавались какие-либо его экземпляры. Их созданию можно воспрепятствовать, написав конструктор, вызываемый по умолчанию, без параметров, который ничего не делает и помечен как private. При отсутствии открытых конструкторов будет невозможно создать экземпляр этого класса.[1]

Использование статических полей

В основном статические переменные класса применяются для отслеживания количества экземпляров класса, существующих в данный момент. Это иллюстрируется примером 4.5.

Пример 4.5. Использование статических полей для подсчета экземпляров

```
using System;

public class Cat
{

     public Cat()
     {
          instances++;
     }

     public static void HowManyCats()
     {
          Console.WriteLine("Кошек принято: {0} ",
                instances);
     }
     private static int instances = 0;
}

public class Tester
{
     static void Main()
     {
           Cat.HowManyCats();
           Cat frisky = new Cat();
           Cat.HowManyCats();
           Cat whiskers = new Cat();
           Cat.HowManyCats();
     }

}
```

[1] Можно создать открытый статический метод, который будет вызывать конструктор и создавать экземпляры класса. Обычно это используют для обеспечения существования только одного экземпляра класса. Подобную конструкцию называют паттерном проектирования Singleton; она описана в классическом труде Э. Гаммы (E. Gamma) и др. «Design Patterns» (Addison Wesley, 1995) (перев. с англ.: «Приемы объектно-ориентированного проектирования. Паттерны проектирования» – СПб: Питер, 2000).

Вывод:

```
Кошек принято: 0
Кошек принято: 1
Кошек принято: 2
```

Класс `Cat` содержит самый минимум. В нем создается и инициализируется (нулем) статическая переменная `instances`. Обратите внимание, что статический элемент считается частью класса, а не элементом экземпляра. Поэтому он не может быть инициализирован компилятором в момент создания экземпляра. Таким образом, явный инициализатор совершенно необходим для статических переменных класса. По мере создания (конструктором) дополнительных экземпляров класса `Cat` счетчик будет увеличиваться.[1]

Уничтожение объектов

В C# предусмотрена сборка мусора, и поэтому отсутствует необходимость в явном деструкторе. Однако если объект работает с неуправляемыми ресурсами, программисту приходится явно освобождать их по завершении работы. Это делается с помощью *деструктора* (*destructor*), вызываемого сборщиком мусора при уничтожении объекта.

Деструктор должен лишь освобождать ресурсы, удерживаемые объектом, и не должен ссылаться на другие объекты. Следует заметить, что если используются только управляемые ресурсы, то деструктор не нужен и не должен реализовываться. Поскольку применение деструктора сопряжено с некоторыми затратами, его необходимо применять лишь к нуждающимся в нем методам (а именно к методам, потребляющим ценные неуправляемые ресурсы).

Статические методы доступа к статическим полям

Весьма нежелательно объявлять данные класса открытыми. Это относится и к статическим переменным класса. Тогда как решение об объявлении статических элементов закрытыми, как это сделано в приведенном выше примере (переменная `instances`), вполне приемлемо. Был создан открытый метод `HowManyCats()`, обеспечивающий доступ к закрытому члену класса. Поскольку метод `HowManyCats()` сам является статическим, у него нет проблем с обращением к статической переменной `instances`.

[1] Однако при их уничтожении счетчик уменьшаться не будет. Поэтому фактически он подсчитывает не количество существующих, а количество порожденных экземпляров класса. – *Примеч. науч. ред.*

Никогда непосредственно не вызывайте деструктор объекта. Сборщик мусора сделает это сам.

Как работают деструкторы

Сборщик мусора ведет список объектов, имеющих деструкторы. Этот список обновляется каждый раз, когда появляется или уничтожается подобный объект.

Когда объект из такого списка попадается сборщику мусора, он помещается в очередь объектов, ожидающих уничтожения. После выполнения деструктора сборщик мусора уничтожает объект, обновляет очередь и список объектов с деструкторами.

Деструктор C#

Синтаксически деструктор в языке C# напоминает деструктор C++, однако работает он совершенно по-другому. Объявляется деструктор с помощью символа «тильда» (~):

```
~MyClass(){}
```

В C# этот синтаксис служит лишь сокращенной записью объявления метода `Finalize()`, связывающей его с базовым классом. Таким образом, запись:

```
~MyClass()
{
   // выполнить действия
}
```

преобразуется компилятором C# в:

```
protected override void Finalize()
{
   try
   {
      // выполнить действия
   }
   finally
   {
      base.Finalize();
   }
}
```

Сравнение деструкторов и метода Dispose()

Нельзя явно вызывать деструктор. Он будет вызван сборщиком мусора. Если программист работает с ценными неуправляемыми ресурсами (например с дескрипторами файлов), которые следует закрывать без промедления, он должен реализовать интерфейс `IDisposable`. (Ин-

терфейсы подробно обсуждаются в главе 8.) Этот интерфейс требует, чтобы его разработчик определил метод под названием Dispose(), производящий освобождение критических ресурсов. Наличие этого метода дает разработчику возможность сообщить компьютеру: «Выполнить это действие прямо сейчас, не дожидаясь вызова деструктора».

Реализуя метод Dispose(), программист должен позаботиться о том, чтобы сборщик мусора не вызвал деструктор соответствующего объекта. Для этого необходимо вызвать статический метод GC.SuppressFinalize(), передав ему указатель this на данный объект. После этого деструктор может вызвать Dispose(). Таким образом, можно написать:

```
using System;
class Testing : IDisposable
{
  bool is_disposed = false;
  protected virtual void Dispose(bool disposing)
  {
    if (!is_disposed) // очищать только один раз!
    {
      if (disposing)
      {
        Console.WriteLine("Not in destructor, OK to reference other objects");
      }
      // очищаем этот объект
      Console.WriteLine("Disposing…");
    }
    this.is_disposed = true;
  }

  public void Dispose()
  {
    Dispose(true);
    // запретить сборщику мусора выполнять финализацию
    GC.SuppressFinalize(this);
  }

  ~Testing()
  {
    Dispose(false);
    Console.WriteLine("In destructor.");
  }
}
```

Реализация метода Close()

Для некоторых объектов предпочтительнее, чтобы разработчики вызывали метод Close(). (Например, для файловых объектов метод Close() привычнее, нежели Dispose().) Реализовать это можно, создав закрытый метод Dispose() и открытый Close() так, чтобы последний вызывал первый.

Оператор using

Поскольку нельзя быть уверенным, что пользователь правильно вызовет метод Dispose(), а процесс завершения является случайным (то есть программист не управляет сборщиком мусора), язык C# предоставляет оператор using, гарантирующий, что метод Dispose() будет вызван как можно раньше. Смысл этого оператора в том, что он объявляет, какие объекты используются, и создает области видимости для них при помощи фигурных скобок. По достижении закрывающей скобки метод Dispose() вызывается автоматически, что проиллюстрировано примером 4.6.

Пример 4.6. Конструкция using

```
using System.Drawing;
class Tester
{
   public static void Main()
   {
      using (Font theFont = new Font("Arial", 10.0f))
      {
         // использовать theFont

      }   // компилятор вызовет Dispose() для theFont

      Font anotherFont = new Font("Courier",12.0f);

      using (anotherFont)
      {
         // использовать anotherFont

      }   // компилятор вызовет Dispose() для anotherFont

   }

}
```

В первой части этого примера с помощью оператора using создается объект Font. По окончании оператора using для объекта Font будет вызван метод Dispose().

Во второй части объект Font создается вне оператора using. Когда возникает необходимость в объекте, он помещается в оператор using, а по окончании оператора снова будет вызван метод Dispose().

Кроме прочего, оператор using обеспечивает защиту от непредвиденных исключений. Метод Dispose() вызывается независимо от того, каким образом управление было передано из оператора using. Все происходит, словно в операторе неявно присутствует блок try-catch-finally. (Подробности см. в разделе «Объекты exception» главы 11.)

Передача параметров

По умолчанию типы значений передаются методам по значению (см. раздел «Аргументы метода» ранее в этой главе). Иными словами,

когда объект передается методу, внутри метода создается временная копия объекта. По окончании работы метода копия уничтожается. Хотя передача по значению вполне приемлема, бывают ситуации, когда лучше передавать объекты по ссылке. Язык C# предоставляет модификатор параметра `ref` для передачи переменных методу по ссылке. Кроме того, существует модификатор `out` для тех случаев, когда переменная с модификатором `ref` должна передаваться без предварительной инициализации. C# поддерживает также модификатор `params`, позволяющий методу принимать переменное количество параметров. Ключевое слово `params` обсуждается в главе 9.

Передача по ссылке

Методы могут возвращать лишь одно значение (хотя оно может быть коллекцией из нескольких значений). Вернемся к классу `Time` и добавим метод `GetTime()`, возвращающий часы, минуты и секунды.

Поскольку три значения возвратить невозможно, поступим следующим образом. Передадим методу три аргумента, позволим ему их изменить, а затем проанализируем результат в вызывающем методе. В примере 4.7 демонстрируется первая версия метода.

Пример 4.7. Возвращение результата через параметры

```
public class Time
{
   // открытые методы
   public void DisplayCurrentTime()
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }

   public int GetHour()
   {
      return Hour;
   }

     public void GetTime(int h, int m, int s)
     {
         h = Hour;
         m = Minute;
         s = Second;
     }

   // конструктор
   public Time(System.DateTime dt)
   {
         Year = dt.Year;
         Month = dt.Month;
         Date = dt.Day;
         Hour = dt.Hour;
```

```
            Minute = dt.Minute;
            Second = dt.Second;
        }

         // закрытые переменные класса
        private int Year;
        private int Month;
        private int Date;
        private int Hour;
        private int Minute;
        private int Second;

    }

    public class Tester
    {
        static void Main()
        {
            System.DateTime currentTime = System.DateTime.Now;
            Time t = new Time(currentTime);
            t.DisplayCurrentTime();

                int theHour = 0;
                int theMinute = 0;
                int theSecond = 0;
                t.GetTime(theHour, theMinute, theSecond);
            System.Console.WriteLine("Текущее время: {0}:{1}:{2}",
                    theHour, theMinute, theSecond);

        }

    }
```

Вывод:

```
11/17/2005 13:41:18
Текущее время: 0:0:0
```

Обратите внимание, что в строке «Текущее время» выводится `0:0:0`. Очевидно, первая версия работает неправильно. Проблема в передаче аргументов. Методу `GetTime()` передаются три аргумента, он изменяет их значения, но при обращении к переданным переменным в методе `Main()` они оказываются неизмененными. Это происходит потому, что `int` является размерным типом, следовательно, параметры передаются по значению. Иными словами, в методе `GetTime()` создаются их копии. Отсюда вывод – следует передавать аргументы по ссылке.

Необходимо внести в код два небольших изменения. Во-первых, параметры в методе `GetTime()` должны быть помечены модификатором `ref` как ссылочные:

```
public void GetTime(ref int h, ref int m, ref int s)
{
     h = Hour;
     m = Minute;
```

```
        s = Second;
    }
```

Во-вторых, в вызове метода GetTime() также надо указать, что аргументы передаются по ссылке:

```
t.GetTime(ref theHour, ref theMinute, ref theSecond);
```

Если не сделать второе изменение и не пометить параметры ключевым словом ref, то компилятор выдаст сообщение о невозможности преобразовать тип int в ref int.

Теперь выводятся корректные результаты. Объявив, что параметры ссылочные, программист инструктирует компилятор передавать их по ссылке. Никакие копии не создаются, а параметр в GetTime() является ссылкой на соответствующую переменную (например, theHour), созданную в методе Main(). Изменение значений в методе GetTime() отражается на переменных метода Main().

Следует помнить, что аргументы с модификатором ref представляют собой ссылки на оригинальное значение. Программист как бы говорит компилятору: «Обработать вон тот объект». Параметры, передаваемые по значению, напротив, являются копиями. Здесь программист говорит: «Обработать *точную копию* объекта».

Возвращение параметров с их инициализацией

В языке C# действует принцип *явной инициализации*, согласно которому все переменные должны быть инициализированы до первого использования в программе. Если в примере 4.7 не инициализировать переменные theHour, theMinute и theSecond до передачи их в качестве аргументов методу GetTime(), то компилятор выдаст сообщения об ошибках. В рассматриваемом примере этим переменным присваиваются нулевые значения:

```
int theHour = 0;
int theMinute = 0;
int theSecond = 0;
t.GetTime( ref theHour, ref theMinute, ref theSecond);
```

Такая инициализация выглядит глупо, поскольку переменные тут же передаются методу GetTime(), где они будут изменены. Если же ее не выполнить, компилятор выдаст сообщения:

```
Use of unassigned local variable 'theHour'
Use of unassigned local variable 'theMinute'
Use of unassigned local variable 'theSecond'
```

Специально для этого случая в языке C# предусмотрен модификатор параметра out. Он снимает требование на обязательную инициализацию аргумента, передаваемого по ссылке. Аргументы метода GetTime()

не передают ему никакой информации, напротив, они являются средством передачи информации из него. Поэтому, пометив их модификатором `out`, программист избавляет себя от необходимости инициализировать соответствующие переменные вне метода. Внутри же метода эти формальные параметры должны получить какие-либо значения до окончания работы метода. Ниже приводится модифицированное объявление параметров метода `GetTime()`:

```
public void GetTime(out int h, out int m, out int s)
{
    h = Hour;
    m = Minute;
    s = Second;
}
```

А вот новый вызов этого метода из `Main()`:

```
t.GetTime( out theHour, out theMinute, out theSecond);
```

Подводя итоги, скажем, что размерные типы передаются методам по значению. Аргументы с модификатором `ref` служат для передачи размерных типов по ссылке. Это позволяет возвращать их измененные значения в вызывающий метод. Аргументы с модификатором `out` применяются только для передачи информации из вызываемого метода. В примере 4.8 приводится программа из примера 4.7, переписанная так, что в ней задействованы все три вида параметров.

Пример 4.8. Входные, выходные и ссылочные параметры

```
public class Time
  {
     // открытые методы доступа
     public void DisplayCurrentTime()
     {
        System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
           Month, Date, Year, Hour, Minute, Second);
     }

     public int GetHour()
     {
        return Hour;
     }

       public void SetTime(int hr, out int min, ref int sec)
       {
           // если переданное время >= 30,
           // инкрементировать минуты и обнулить секунды,
           // в противном случае оставить все как есть
           if (sec >= 30)
           {
               Minute++;
               Second = 0;
           }
```

```
            Hour = hr; // присвоить переданное значение

            // возвратить минуты и секунды
            min = Minute;
            sec = Second;
        }

    // конструктор
    public Time(System.DateTime dt)
    {

        Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;
    }

     // закрытые переменные класса
    private int Year;
    private int Month;
    private int Date;
    private int Hour;
    private int Minute;
    private int Second;

}

public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time(currentTime);
        t.DisplayCurrentTime();

            int theHour = 3;
            int theMinute;
            int theSecond = 20;

            t.SetTime(theHour, out theMinute, ref theSecond);
            System.Console.WriteLine(
             "Сейчас: {0} минут и {1} секунд",
                theMinute, theSecond);
            theSecond = 40;
            t.SetTime(theHour, out theMinute, ref theSecond);
            System.Console.WriteLine("Сейчас: " +
               "{0} минут и {1} секунд",
                   theMinute, theSecond);

    }

}
```

Вывод:

```
11/17/2005 14:6:25
Сейчас: 6 минут и 25 секунд
Сейчас: 7 минут и 0 секунд
```

Метод `SetTime()` слегка надуманный, но зато иллюстрирует все три вида параметров. Аргумент `theHour` передается по значению. Его единственная задача – установить переменную `Hour` в нужное значение, и никакую выходную информацию он в себе не несет.

Передаваемый по ссылке аргумент `theSecond` используется для установки значения внутри метода. Если этот параметр больше или равен 30, то переменная `Second` сбрасывается в ноль, а переменная `Minute` увеличивается на 1.

Наконец, аргумент `theMinute` передается методу для того, чтобы получить в нем значение переменной `Minute` и возвратить его. Поэтому он помечен модификатором `out`.

Безусловно, следует инициализировать переменные `theHour` и `theSecond`; их значения нужны вызываемому методу. Зато нет необходимости инициализировать `theMinute`, поскольку эта переменная служит выходным аргументом, который лишь возвращает значение из метода. Правила, которые сначала выглядели произвольно и неоправданно установленными, оказались вполне разумными. Переменные нужно инициализировать, только когда их начальные значения имеют смысл.

Перегрузка методов и конструкторов

Нередко требуется, чтобы одно и то же имя было сразу у нескольких функций. Самым распространенным случаем является наличие нескольких конструкторов. В примерах, приводимых до сих пор, конструктор принимал один параметр, объект `DateTime`. Было бы удобно устанавливать в новых объектах `Time` произвольное время, передавая им значения года, месяца, числа, часов, минут и секунд. Было бы еще удобнее, если бы одни разработчики могли пользоваться одним конструктором, а другие – другим. Такое понятие, как перегрузка функций, существует именно для этих ситуаций.

Сигнатура (*signature*) метода определяется его именем и списком параметров. Два метода отличаются по сигнатурам, если у них разные имена или списки параметров. Списки параметров могут отличаться по количеству или типам параметров. Например, в следующем фрагменте программы первый метод отличается от второго количеством параметров, а второй от третьего – их типами.

```
void myMethod(int p1);
void myMethod(int p1, int p2);
void myMethod(int p1, string s1);
```

Класс может иметь любое количество методов, если их сигнатуры отличаются друг от друга.

В примере 4.9 демонстрируется класс Time с двумя конструкторами, один из которых принимает объект DateTime, а второй – шесть целых чисел.

Пример 4.9. Перегрузка конструктора

```
public class Time
{
   // открытые методы доступа
   public void DisplayCurrentTime()
   {
      System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second);
   }

   // конструкторы
   public Time(System.DateTime dt)
   {
   Year =      dt.Year;
   Month =     dt.Month;
   Date =      dt.Day;
   Hour =      dt.Hour;
   Minute =    dt.Minute;
   Second =    dt.Second;
   }

    public Time(int Year, int Month, int Date,
         int Hour, int Minute, int Second)
      {
         this.Year =     Year;
         this.Month =    Month;
         this.Date =     Date;
         this.Hour =     Hour;
         this.Minute =   Minute;
         this.Second =   Second;
     }

    // закрытые переменные класса
   private int Year;
   private int Month;
   private int Date;
   private int Hour;
   private int Minute;
   private int Second;

}

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

```
        System.DateTime currentTime = System.DateTime.Now;

        Time t = new Time(currentTime);
        t.DisplayCurrentTime();

        Time t2 = new Time(2005,11,18,11,03,30);
        t2.DisplayCurrentTime();

    }

}
```

Как видно из примера, у класса `Time` два конструктора. Если бы сигнатура состояла только из имени функции, компилятор не знал бы, какой конструктор вызывать при создании объектов `t1` и `t2`. Однако поскольку сигнатура включает в себя и типы аргументов, компилятор в состоянии сопоставить вызов конструктора для `t1` с описанием конструктора, требующего объект `DateTime`. Аналогичным образом компилятор способен связать вызов конструктора объекта `t2` с конструктором, сигнатура которого содержит список из шести аргументов с типом `int`.

Перегружая метод, программист должен изменить сигнатуру (то есть имя метода, количество параметров или их тип). Можно, к тому же, изменить и тип возвращаемого значения, но это не обязательно. Изменение возвращаемого типа не ведет к перегрузке метода, а наличие двух методов с одинаковыми сигнатурами, но разными возвращаемыми типами вызовет ошибку на этапе компиляции. Сказанное иллюстрируется примером 4.10.

Пример 4.10. Изменение возвращаемого типа перегруженных методов

```
public class Tester
{
    private int Triple(int val)
    {
        return 3 * val;
    }

    private long Triple (long val)
    {
        return 3 * val;
    }

    public void Test()
    {
        int x = 5;
        int y = Triple(x);
        System.Console.WriteLine("x: {0}  y: {1}", x, y);

        long lx = 10;
        long ly = Triple(lx);
        System.Console.WriteLine("lx: {0}  ly: {1}", lx, ly);

    }
    static void Main()
    {
```

```
        Tester t = new Tester();
        t.Test();
    }
}
```

В этом примере класс Tester перегружает метод Triple(), один раз с целочисленным параметром, а другой – с параметром типа long. Возвращаемые типы у методов Triple() разные. Хотя это не обязательно, в данном случае разница возвращаемых типов очень удобна.

Инкапсуляция данных в свойствах

Свойства обеспечивают пользователю возможность читать состояние класса, как если бы он обращался непосредственно к полям класса, причем фактически этот доступ реализуется с помощью метода класса.

Это идеальный вариант. Пользователь хочет обладать правом прямого доступа к состоянию объекта, не желая иметь дело с методами. Разработчик класса, со своей стороны, стремится скрыть внутреннее состояние класса в членах класса и предоставляет клиенту косвенный доступ, через методы.

Отделив состояние класса от метода, имеющего доступ к этому состоянию, разработчик получает свободу менять внутреннее состояние объекта как пожелает. Когда класс Time создается в первый раз, значение Hour может храниться в переменной класса. Предположим, через некоторое время разработчик изменяет класс таким образом, что значение Hour вычисляется или, например, берется из базы данных. Если бы пользователь имел доступ к исходной переменной Hour, после такого изменения его программа перестала бы работать. Заставляя клиента применять метод (или свойство), программист может вносить изменения во внутреннюю структуру класса Time, не приводя при этом к неработоспособности использовавших его программ.

Свойства удовлетворяют обоим требованиям. Они предоставляют пользователю простой интерфейс, поскольку выглядят как переменные класса. С другой стороны, они реализованы в виде методов и скрывают данные в соответствии с требованиями объектно-ориентированного программирования. Пример 4.11 иллюстрирует эти положения.

Пример 4.11. Использование свойства

```
public class Time
{
    // открытые методы доступа
    public void DisplayCurrentTime()
    {

        System.Console.WriteLine(
            "Time\t: {0}/{1}/{2} {3}:{4}:{5}",
            month, date, year, hour, minute, second);
```

```
   }

   // конструкторы
   public Time(System.DateTime dt)
   {
      year =      dt.Year;
      month =     dt.Month;
      date =      dt.Day;
      hour =      dt.Hour;
      minute =    dt.Minute;
      second =    dt.Second;
   }

     // создание свойства
     public int Hour
     {
         get
         {
             return hour;
         }

         set
         {
             hour = value;
         }
     }

    // закрытые переменные класса
   private int year;
   private int month;
   private int date;
   private int hour;
   private int minute;
   private int second;
}

public class Tester
{
   static void Main()
   {
      System.DateTime currentTime = System.DateTime.Now;
      Time t = new Time(currentTime);
      t.DisplayCurrentTime();

         int theHour = t.Hour;
         System.Console.WriteLine("\nПолученное значение  hour: {0}\n",
            theHour);
         theHour++;
         t.Hour = theHour;
         System.Console.WriteLine("Измененное значение hour: {0}\n",
            theHour);
   }
}
```

Чтобы объявить свойство, напишите его тип и имя и поставьте пару фигурных скобок. Внутри скобок можно объявить *процедуры доступа* (*accessor*) к свойству, реализующие чтение и изменение его состояния. У них не должно быть явных параметров, хотя, как будет показано ниже, процедура доступа `set()` имеет неявный параметр `value`.

В примере 4.11 приводится свойство `Hour`. В его объявлении указаны две процедуры доступа: `get` и `set`.

```
public int Hour
{
    get
    {
        return hour;
    }

    set
    {
        hour = value;
    }
}
```

Каждая из этих процедур доступа имеет тело, в котором и выполняется вся работа по чтению или изменению значения свойства. В принципе, значение свойства может храниться в базе данных (тогда в теле процедуры доступа должно выполняться обращение к этой базе данных) или просто в закрытой переменной класса:

```
private int hour;
```

Процедура доступа get

Тело процедуры доступа `get` похоже на метод класса, возвращающий объект того же типа, что и свойство. В рассматриваемом примере процедура доступа свойства `Hour` аналогична методу, возвращающему значение типа `int`. Процедура доступа `get` возвращает значение закрытой переменной класса, в которой хранится значение свойства.

```
get
{
    return hour;
}
```

Здесь возвращается локальная переменная типа `int`, однако ничто не мешает считывать значение из базы данных или вычислять его.

Всякий раз, когда в программе производится обращение к свойству (за исключением случаев присваивания ему значения), вызывается процедура доступа `get`, которая читает значение этого свойства:

```
Time t = new Time(currentTime);
int theHour = t.Hour;
```

В этом примере значение свойства Hour объекта Time читается вызовом процедуры доступа get, а затем присваивается локальной переменной.

Процедура доступа set

Процедура доступа set изменяет значение свойства и аналогична методу, не возвращающему значения. Когда программист определяет процедуру доступа set, он должен использовать ключевое слово value, представляющее собой аргумент, значение которого передается и сохраняется в свойстве.

```
set
{
     hour = value;
}
```

Здесь, как и раньше, для хранения значения свойства применяется закрытая переменная класса. Впрочем, процедура доступа set могла бы с таким же успехом записывать значение в базу данных или обновлять другие переменные класса, если возникнет необходимость.

Когда в программе свойству присваивается какое-либо значение, автоматически вызывается процедура доступа set, а неявный параметр value получает значение, присваиваемое переменной:

```
theHour++;
t.Hour = theHour;
```

Преимущество такого подхода состоит в том, что пользователь взаимодействует со свойствами непосредственно, а программист не поступается принципами сокрытия данных и инкапсуляции, святыми для объектно-ориентированного программирования.

Поля, предназначенные только для чтения

Предположим, для каких-то целей понадобилась версия класса Time, предоставляющая открытые статические значения текущего времени и даты. В примере 4.12 дано простое решение этой задачи.

Пример 4.12. Использование статических открытых констант

```
public class RightNow
{
   static RightNow()
   {
      System.DateTime dt = System.DateTime.Now;
      Year =      dt.Year;
      Month =     dt.Month;
      Date =      dt.Day;
      Hour =      dt.Hour;
```

```
        Minute =    dt.Minute;
        Second =    dt.Second;
    }

    // открытые переменные класса
    public static int Year;
    public static int Month;
    public static int Date;
    public static int Hour;
    public static int Minute;
    public static int Second;
}

public class Tester
{
    static void Main()
    {
        System.Console.WriteLine ("Год: {0}",
               RightNow.Year.ToString());
        RightNow.Year = 2006;
        System.Console.WriteLine ("Год: {0}",
        RightNow.Year.ToString());
    }
}
```

Вывод:

```
Год: 2005
Год: 2006
```

Все будет работать хорошо, пока кто-нибудь не придет и не изменит одно из значений. Как видно из примера, значение `RightNow.Year` можно изменить, например, на 2003. Очевидно, это не то, что требовалось.

Чтобы добиться желаемого результата, хотелось бы пометить статические значения как постоянные, но это невозможно, потому что они не инициализируются до выполнения статического конструктора. Именно для такого случая в языке C# предусмотрено ключевое слово `readonly`. Если изменить описания переменных класса на следующие:

```
public static readonly int Year;
public static readonly int Month;
public static readonly int Date;
public static readonly int Hour;
public static readonly int Minute;
public static readonly int Second;
```

а затем закомментировать повторное присваивание в методе `Main()`:

```
// RightNow.Year = 2002; // ошибка!
```

то программа будет скомпилирована без ошибок и будет работать, как задумано.

5

Наследование и полиморфизм

В предыдущей главе было показано, как создавать новые типы, объявляя классы. В настоящей главе рассматриваются отношения между объектами в реальном мире и демонстрируются способы моделирования их в программе. Обсуждается понятие *специализации* (*specialization*), которое реализуется в C# с помощью механизма *наследования* (*inheritance*). Кроме того, здесь разъясняется, как экземпляры специализированных классов могут рассматриваться в качестве экземпляров более общих классов, – явление, называемое *полиморфизмом* (*polymorphism*). Глава заканчивается рассмотрением *изолированных* (*sealed*) классов, которые не могут быть подвергнуты специализации, *абстрактных* (*abstract*) классов, которые только для того и существуют, чтобы их специализировали, а также класса `Object`, являющегося корневым для всех остальных классов.

Специализация и обобщение

Классы и их экземпляры (объекты) существуют не в вакууме. Они существуют в сложной системе взаимозависимостей и отношений, совсем как люди живут в обществе со сложной системой взаимоотношений и социальных уровней.

Отношение *является* (чем-то или кем-то) – это отношение *специализации*. Когда говорят, что собака *является* млекопитающим, подразумевают, что собака – это специализированный вид млекопитающих. Она обладает всеми характеристиками млекопитающих (живорождение, вскармливание детенышей молоком, наличие шерсти), но конкретизирует (специализирует) эти характеристики до характеристик вида *canine domesticus*. Кошка тоже млекопитающее. Наблюдатель вправе

ожидать, что у нее есть ряд характеристик таких же, как у собаки (общих для всех млекопитающих), но присутствуют и другие, нашедшие свою специализацию в кошке.

Специализация и обобщение – отношения взаимные и иерархические. Они взаимны, поскольку специализация является обратной стороной обобщения. Так, собака и кошка являются специализацией для млекопитающего, а млекопитающее – обобщение для собаки и кошки.

Эти отношения иерархичны, поскольку они порождают дерево отношений, в котором специализированные типы ответвляются от более общих типов. При движении вверх по дереву достигается большее обобщение. Двигаясь вверх к млекопитающему, можно обобщить, что собаки, кошки и лошади являются живородящими. Двигаясь вниз, можно специализировать. Таким образом, кошка специализирует млекопитающее тем, что имеет когти (характеристика) и мяукает (поведение).

Аналогичным образом, если сказать, что список (`ListBox`) и кнопка (`Button`) *являются* окнами (`Window`), то сказанное будет означать наличие в этих элементах управления свойств и поведения, присущих типу `Window`. Другими словами, `Window` обобщает характеристики `ListBox` и `Button`, присущие обоим объектам, а каждый из них специализирует собственные храктеристики и свое поведение.

Об унифицированном языке моделирования (UML)

Язык UML (Unified Modeling Language, унифицированный язык моделирования) является стандартным языком для описания системы или коммерческой деятельности. Та часть UML, которая может оказаться полезной для данной главы, является набором схем, документирующих отношения между классами.

В UML классы представлены в виде прямоугольников. Название класса написано в верхней части прямоугольника, а методы и члены перечисляются под именем (но это необязательно).

Пример моделирования отношения на языке UML показан на рис. 5.1. Обратите внимание, что стрелки ведут от специализированного класса к более общему.

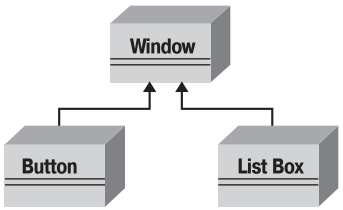

Рис. 5.1. *Отношение «является»*

Распространенным приемом разработки приложений является поиск общих функциональных особенностей у двух классов и выделение этих черт в общий базовый класс. Такой подход позволяет повторно использовать одни и те же фрагменты программы и облегчает ее сопровождение.

Представим, например, что разрабатывается ряд объектов, изображенных на рис. 5.2.

Поработав некоторое время с объектами `RadioButton`, `CheckBox` и `Command`, программист обнаруживает, что некоторые их характеристики и особенности поведения являются общими для всех, но более специализированными, чем у `Window`. Эти общие черты можно выделить в базовый класс, назовем его `Button`, и реорганизовать схему наследования, как показано на рис. 5.3. Вот пример того, как обобщение используется в объектно-ориентированном программировании.

Эта UML-диаграмма отражает отношения между выделенными классами и демонстрирует, что `ListBox` и `Button` произведены от `Window`, а

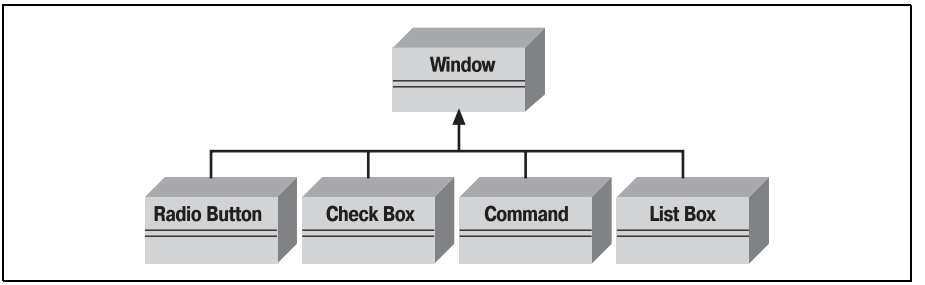

Рис. 5.2. *Объекты, производные от Window*

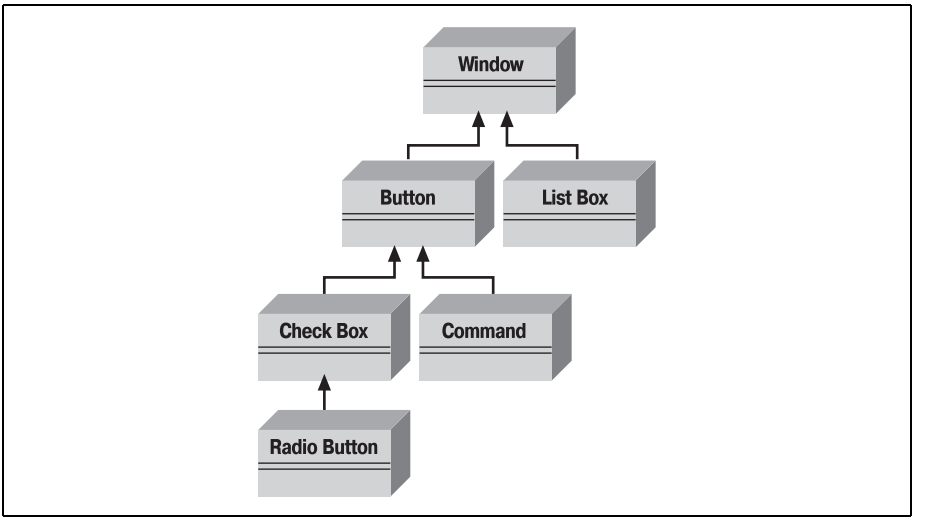

Рис. 5.3. *Уточненная иерархическая структура*

Button, в свою очередь, специализируется классами CheckBox и Command. Наконец, RadioButton производится от CheckBox. Таким образом, можно сказать, что переключатель (RadioButton) является флажком (CheckBox), который, в свою очередь, является кнопкой (Button), а кнопка является окном (Window).

Здесь предложена не единственная и даже не самая лучшая организация этих объектов, однако она может послужить отправной точкой для понимания того, как эти типы (классы) соотносятся друг с другом.

Хотя сказанное выше, возможно, отражает иерархическую структуру объектов, автор скептически относится к любой системе, в которой модель не соответствует реальным ощущениям пользователя. Прежде чем написать, что переключатель является флажком, автор какое-то время сомневался в правомочности такого заявления. Быть может, лучше сказать, что переключатель является разновидностью флажка, в которой реализуется идея взаимоисключающего выбора. То есть первоначальное утверждение («переключатель – это флажок») звучит с натяжкой и, быть может, свидетельствует о несовершенстве модели.

Наследование

В языке C# отношение специализации реализуется, как правило, с помощью наследования. Это не единственный, но самый распространенный и естественный способ реализации такого отношения.

Утверждение, что окно со списком (ListBox) наследуется от (или является потомком) окна (Window), равносильно утверждению, что окно со списком – это специализация окна. В этом контексте Window называется *базовым* классом, а ListBox – *производным* классом. Иными словами, характеристики и поведение класса ListBox произведены от класса Window, а затем специализированы под нужды ListBox.

Реализация наследования

В C# производный класс создается постановкой двоеточия после имени производного класса и указанием имени базового класса:

```
public class ListBox : Window
```

В этой строке объявляется новый класс, ListBox, производный от класса Window. Двоеточие можно озвучить как «произведен от». Производный класс наследует все элементы базового – как переменные, так и методы. При этом производный класс имеет право реализовать собственную версию метода базового класса. Он делает это, помечая новый метод ключевым словом new. (Ключевое слово new обсуждается также в разделе «Создание версий с помощью ключевых слов new и override»

далее в этой главе.) Это ключевое слово означает, что производный класс намеренно заменил метод базового класса на свой собственный, как показано в примере 5.1.

Пример 5.1. Применение производного класса

```
using System;

public class Window
{
   // конструктор принимает два целых числа -
   // координаты на экране
   public Window(int top, int left)
   {
      this.top = top;
      this.left = left;
   }

   // имитация рисования окна
   public void DrawWindow()
   {
      Console.WriteLine("Рисуем окно с координатами {0}, {1}",
         top, left);
   }

   // эти элементы закрыты и потому невидимы
   // для методов производного класса,
   // что обсуждается далее в этой главе
   private int top;
   private int left;
}

// ListBox является потомком Window
public class ListBox : Window
{
   // в конструктор добавляется формальный параметр
   public ListBox(
      int top,
      int left,
      string theContents):
      base(top, left)  // вызов конструктора базового класса
   {
      mListBoxContents = theContents;
   }

   // новая версия (с ключевым словом new), поскольку
   // в производном методе меняется поведение класса
   public new void DrawWindow()
   {
      base.DrawWindow();  // вызов метода базового класса
      Console.WriteLine ("Записываем строку в список: {0}",
         mListBoxContents);
   }
   private string mListBoxContents;  // новая переменная класса
```

```
}

public class Tester
{
   public static void Main()
   {
      // создать экземпляр базового класса
      Window w = new Window(5,10);
      w.DrawWindow();

      // создать экземпляр производного класса
      ListBox lb = new ListBox(20,30,"Hello world");
      lb.DrawWindow();
   }
}
```

Вывод:

```
Рисуем окно с координатами 5, 10
Рисуем окно с координатами 20, 30
Записываем строку в список: Hello world
```

Пример 5.1 начинается с объявления базового класса `Window`. В этом классе реализованы конструктор и простой метод, имитирующий рисование. Кроме того, в нем определены две закрытые переменные, `top` и `left`.

Вызов конструктора базового класса

В примере 5.1 новый класс `ListBox` является потомком класса `Window` и имеет собственный конструктор, который принимает три параметра. Конструктор `ListBox` вызывает конструктор базового класса, поскольку после списка его параметров стоит двоеточие, а за ним – ключевое слово `base`, обозначающее базовый класс:

```
public ListBox(
    int theTop,
    int theLeft,
    string theContents):
    base(theTop, theLeft)  // вызов конструктора базового класса
```

Из-за того что классы не могут наследовать конструкторы, производный класс должен реализовать собственный конструктор, а конструктор базового класса можно использовать, лишь вызывая его явно.

Кроме того, обратите внимание, как в примере 5.1 реализуется новая версия метода `DrawWindow()`:

```
public new void DrawWindow()
```

Здесь ключевое слово `new` указывает, что программист сознательно создает новую версию метода в производном классе.

Если у базового класса есть открытый конструктор, используемый по умолчанию, производный конструктор не обязан явно вызывать конструктор своего базового класса – по умолчанию конструктор будет вызван неявно. Однако если базовый класс не имеет конструктора, используемого по умолчанию, каждый производный конструктор *обязан* явно вызывать один из конструкторов базового класса с помощью ключевого слова `base`.

Как было сказано в главе 4, если никакой конструктор не объявлен, компилятор создаст конструктор по умолчанию. По умолчанию будет вызываться конструктор без формальных параметров независимо от того, написан он программистом или предоставлен компилятором. Обратите внимание, что если программист все-таки написал конструктор (с параметрами или без), компилятор не станет создавать конструктор, используемый по умолчанию.

Вызов методов базового класса

В примере 5.1 метод `DrawWindow()` класса `ListBox` замещает собой метод базового класса. Когда метод `DrawWindow()` вызывается для объекта типа `ListBox`, это будет метод `ListBox.DrawWindow()`, но не `Window.DrawWindow()`. Впрочем, обратим внимание, что метод `ListBox.DrawWindow()` может вызвать метод `DrawWindow()` своего базового класса с помощью следующей конструкции:

```
base.DrawWindow();  // вызов метода базового класса
```

(Ключевое слово `base` обозначает базовый класс для текущего объекта.)

Управление доступом

Область видимости класса и его элементов может быть ограничена с помощью модификаторов прав доступа `public`, `private`, `protected`, `internal` и `protected internal`. (Модификаторы прав доступа рассматривались в главе 4.)

Как читатель уже знает, модификатор `public` делает элемент класса доступным для методов других классов, а `private` означает, что элемент виден только методам того же класса. Ключевое слово `protected` расширяет область видимости, открывая элемент объектам производных классов, в то время как слово `internal` открывает его методам любого класса из той же *сборки*.[1]

---

[1] Сборка (обсуждавшаяся в главе 1) – это основная единица для совместного и повторного использования кода в среде CLR (логическая DLL). В типичном случае сборка является набором физических файлов, хранящихся в одном каталоге, включающих в себя все ресурсы (например, изображения), необходимые выполняемому файлу, а также код на промежуточном языке (IL) и метаданные для программы.

Пара ключевых слов `internal protected` предоставляет доступ элементам той же сборки (`internal`) *или* элементам производных классов (`protected`). Этот модификатор доступа имеет семантику `internal` *или* `protected`.

Классам, как и их элементам, может быть назначен любой из этих уровней доступа. Если для элемента указан иной модификатор прав доступа, чем для класса, приоритет у более строгого модификатора. Например, если класс `myClass` определен как

```
public class myClass
{
   // ...
   protected int myValue;
}
```

то уровень доступа переменной `myValue` будет `protected`, хотя сам класс является открытым. *Открытый* (*public*) *класс* – это класс, доступный любому классу, желающему взаимодействовать с ним. Если же класс создается только как вспомогательный для других классов сборки, то его лучше пометить модификатором `internal`, а не `public`.

Полиморфизм

У наследования есть две важных особенности. Первая – многократное использование кода. Создав класс `ListBox`, программист может использовать часть возможностей базового класса (`Window`).

Однако второй аспект наследования, возможно, является более ценным. Это *полиморфизм* (*polymorphism*). *Poly* означает «много», *morph* – «форма». Таким образом, если не вдаваться в детали, полиморфизм означает возможность использования многих форм одного типа.

Когда телефонная станция посылает сигнал абоненту, она не имеет представления о телефонном аппарате абонента. Это может быть старенький телефон производства Western Electric с электромеханическим звонком, а может быть современный аппарат, воспроизводящий цифровую музыку.

Телефонная станция знает лишь, что есть «базовый тип» phone (телефон) и считает, что каждый «экземпляр» этого типа способен издавать звуковой сигнал. Когда станция велит телефону зазвонить, она ожидает, что он «сделает то, что требуется». Таким образом, телефонная станция обращается к вашему телефону полиморфно.

Создание полиморфных типов

Поскольку окно со списком (`ListBox`) *является* окном (`Window`) и кнопка (`Button`) также *является* окном, необходимо, чтобы программист мог обратиться к любому из этих двух типов в ситуациях, когда речь идет

о типе Window. Например, экранная форма может содержать коллекцию всех своих элементов типа Window. При своем выводе на экран она велит всем экземплярам Window нарисовать самих себя. Форме нет никакого дела до того, списки это или кнопки. Она просто проходит по коллекции элементов и велит каждому «нарисоваться». Короче говоря, форма обращается с объектами Window полиморфно.

Создание полиморфных методов

Чтобы создать метод, поддерживающий полиморфизм, достаточно лишь пометить его ключевым словом virtual в базовом классе. Например, чтобы сделать полиморфным метод DrawWindow() класса Window в примере 5.1, просто добавьте в объявление слово virtual:

```
public virtual void DrawWindow()
```

Теперь каждый производный класс волен реализовать собственную версию метода DrawWindow(). Для этого программист переопределяет виртуальный метод базового класса, используя ключевое слово override в описании производного класса, а затем пишет новый код метода.

В следующем отрывке из примера 5.2 (который появится несколько позже) класс ListBox наследуется от класса Window и реализует собственную версию метода DrawWindow():

```
public override void DrawWindow()
{
   base.DrawWindow();  // вызов метода базового класса
   Console.WriteLine ("Запись строки в список: {0}",
      listBoxContents);
}
```

Ключевое слово override сообщает компилятору, что класс сознательно переопределяет работу метода DrawWindow(). Аналогичным образом переопределяется этот метод в классе Button, тоже производном от Window.

В теле программы из примера 5.2, в первую очередь, создаются три объекта, Window, ListBox и Button. Затем для каждого вызывается метод DrawWindow():

```
Window win = new Window(1,2);
ListBox lb = new ListBox(3,4,"Изолированный список");
Button b = new Button(5,6);
win.DrawWindow();
lb.DrawWindow();
b.DrawWindow();
```

Этот код работает так, как и следовало ожидать. В каждом случае вызывается правильный метод DrawWindow(). До сих пор никакого полиморфизма не было. Чудеса начнутся, когда будет создан массив объектов Window. Поскольку тип ListBox *является* типом Window, можно по-

местить объект ListBox в массив объектов типа Window. По сходной причине в этот массив можно поместить объект Button:

```
Window[] winArray = new Window[3];
winArray[0] = new Window(1,2);
winArray[1] = new ListBox(3,4,"Список в массиве");
winArray[2] = new Button(5,6);
```

Что произойдет при вызове метода DrawWindow() для каждого из этих объектов?

```
for (int i = 0;i < 3; i++)
{
   winArray[i].DrawWindow();
}
```

Компилятор знает, что у него есть три объекта Window и что для каждого вызван метод DrawWindow(). Если бы этот метод не был предварительно помечен как виртуальный, все три раза был бы вызван метод DrawWindow() класса Window. Однако поскольку он все-таки объявлен с ключевым словом virtual и поскольку производные классы переопределили его, при вызове метода для очередного элемента массива компилятор распознает, какие объекты фактически будут задействованы на этапе выполнения (а именно Window, ListBox и Button), и вызовет для каждого соответствующий метод. В этом суть полиморфизма. Полный текст рассматриваемой программы приводится в примере 5.2.

В этой программе используется массив, представляющий собой коллекцию однотипных объектов. Обращение к элементам массива осуществляется с помощью оператора индексирования:

```
// установить значение элемента
// с номером 5
MyArray[5] = 7;
```

Первый элемент любого массива имеет индекс 0. Использование массива в данном примере должно быть интуитивно понятно. Более подробно массивы рассматриваются в главе 9.

Пример 5.2. Использование виртуальных методов

```
using System;

public class Window
{
   // конструктор принимает два целых числа -
   // координаты на экране
   public Window(int top, int left)
   {
      this.top = top;
```

```
        this.left = left;
    }

    // имитация рисования окна
    public virtual void DrawWindow()
    {
        Console.WriteLine("Window: рисуем окно с координатами {0}, {1}",
            top, left);
    }

    // эти элементы защищенные и поэтому видимы
    // для методов производного класса,
    // что обсуждается далее в этой главе
    protected int top;
    protected int left;

}

// ListBox является потомком Window
public class ListBox : Window
{
    // конструктор добавляет параметр
    public ListBox(
        int top,
        int left,
        string contents):
        base(top, left)  // вызов конструктора базового класса
    {

        listBoxContents = contents;
    }

    // перегруженная версия (с ключевым словом override), поскольку
    // в производном методе меняется поведение класса
    public override void DrawWindow()
    {
        base.DrawWindow();  // вызов метода базового класса
        Console.WriteLine ("Запись строки в список: {0}",
            listBoxContents);
    }

    private string listBoxContents;  // новая переменная класса
}

public class Button : Window
{
    public Button(
        int top,
        int left):
        base(top, left)
    {
    }

    // перегруженная версия (с ключевым словом override), поскольку
    // в производном методе меняется поведение класса
```

```
    public override void DrawWindow()
    {
        Console.WriteLine("Рисуем кнопку с координатами {0}, {1}\n",
            top, left);
    }
}

public class Tester
{
    static void Main()
    {
        Window win = new Window(1,2);
        ListBox lb = new ListBox(3,4,"Изолированный список");
        Button b = new Button(5,6);
        win.DrawWindow();
        lb.DrawWindow();
        b.DrawWindow();

        Window[] winArray = new Window[3];
        winArray[0] = new Window(1,2);
        winArray[1] = new ListBox(3,4,"Список в массиве");
        winArray[2] = new Button(5,6);

        for (int i = 0;i < 3; i++)
        {
            winArray[i].DrawWindow();
        }
    }
}
```

Вывод:

```
Window: рисуем окно с координатами 1, 2
Window: рисуем окно с координатами 3, 4
Запись строки в список: Изолированный список
Рисуем кнопку с координатами 5, 6

Window: рисуем окно с координатами 1, 2
Window: рисуем окно с координатами 3, 4
Запись строки в список: Список в массиве
Рисуем кнопку с координатами 5, 6
```

Обратите внимание, что во всем примере новые переопределяемые методы помечены ключевым словом `override`:

```
public override void DrawWindow()
```

Теперь компилятор знает, что к объектам надо относиться полиморфно и вызывать переопределенные методы. Компилятор несет ответственность за отслеживание реального типа объекта и за выполнение позднего связывания, то есть за вызов метода `ListBox.DrawWindow()`, когда переменная типа `Window` в действительности указывает на объект `ListBox`.

Вниманию программистов, пишущих на языке C++! Необходимо явно помечать ключевым словом `override` объявление каждого метода, который переопределяет виртуальный.

Создание версий с помощью ключевых слов new и override

В языке C# решение программиста переопределить виртуальный метод объявляется ключевым свойством `override`. Это обстоятельство позволяет выпускать новые версии кода, поскольку изменения в базовом классе не приведут к сбоям в работе производных классов. Требование указывать ключевое слово `override` снимает потенциальные проблемы.

Обсудим это подробнее. Пусть базовый класс `Window` из предыдущего примера разработан в фирме A. Предположим также, что классы `ListBox` и `RadioButton` были написаны программистами из фирмы B, которые в качестве базового класса взяли класс `Window`, приобретенный у фирмы A. Программисты фирмы B не могут или почти не могут влиять на разработку класса `Window`, в том числе на исправления, вносимые фирмой A.

И вот один из программистов фирмы B добавляет в класс `ListBox` метод `Sort()`:

```
public class ListBox : Window
{
   public virtual void Sort() {...}
}
```

Дела идут хорошо, пока компания A, автор класса `Window`, не выпустит версию 2 этого класса, содержащую метод `Sort()`:

```
public class Window
{
   // ...
   public virtual void Sort() {...}
}
```

В других объектно-ориентированных языках (например C++) новый виртуальный метод `Sort()` класса `Window` будет действовать как базовый для виртуального метода `Sort()` класса `ListBox`. Компилятор вызовет метод `Sort()` класса `ListBox`, когда программист вознамерится вызвать метод `Sort()` класса `Window`. В языке Java, если метод `Sort()` класса `Window` возвращает другой тип данных, то загрузчик классов сочтет `Sort()` в `ListBox` недопустимым переопределением метода и откажется его загружать.

Язык C# предотвращает возникновение подобной неразберихи. В нем виртуальная функция всегда считается корнем виртуальной цепочки.

Иными словами, когда компилятор C# обнаруживает виртуальный метод, он не выполняет поиск на более высоких уровнях иерархической структуры наследования. Если в классе Window появится новая виртуальная функция Sort(), поведение класса ListBox на этапе выполнения не изменится.

Однако если ListBox скомпилировать заново, то будет сгенерировано следующее предупреждение:

```
...\class1.cs(54,24): warning CS0114: 'ListBox.Sort()' hides
inherited member 'Window.Sort()'.
To make the current member override that implementation,
add the override keyword. Otherwise add the new keyword.
```

```
(... 'ListBox.Sort()' скрывает наследуемый элемент 'Window.Sort()'.
Чтобы текущий элемент переопределил эту реализацию, добавьте ключевое слово
override. В противном случае добавьте ключевое слово new)
```

Чтобы избавиться от этого предупреждения, программист должен явно сформулировать свои намерения. Он может пометить метод Sort() класса ListBox словом new, и тогда этот метод *не будет* переопределять виртуальный метод класса Window:

```
public class ListBox : Window
{
   public new virtual void Sort() {...}
```

Тогда предупреждение больше не появится. Если программист, напротив, хочет переопределить метод класса Window, ему будет достаточно явно сообщить об этом с помощью ключевого слова override:

```
public class ListBox : Window
{
   public override void Sort() {...}
```

У читателя может возникнуть искушение добавить ключевое слово new ко всем виртуальным методам, чтобы вообще избежать появления подобного предупреждения. Это крайне неудачная идея. Когда в коде появляется слово new, оно документирует факт создания новой версии кода. Оно указывает потенциальному пользователю базового класса, что этот класс на самом деле не переопределяется. Бессистемное применение слова new подрывает принципы такого документирования. В конце концов, предупреждение для того и существует, чтобы помочь программисту разобраться в ситуации.

Абстрактные классы

Каждому подклассу класса Window *следовало бы* реализовать собственный метод DrawWindow(), но он не обязан делать это. Если требуется,

чтобы каждый подкласс непременно реализовал какой-либо метод базового класса, программист должен определить этот метод как *абстрактный* (*abstract*).

Абстрактный метод не имеет реализации. Он лишь создает имя и сигнатуру метода, который должен быть реализован во всех производных классах. Более того, если в каком-то классе один или несколько методов обозначены как абстрактные, то и весь класс становится абстрактным.

Абстрактные классы устанавливают базу для производных классов, но создать объект абстрактного класса нельзя. Объявив абстрактным какой-либо метод класса, программист запрещает создание экземпляров этого класса.

То есть объявив, что метод `DrawWindow()` класса `Window` является абстрактным, программист по-прежнему может производить классы от `Window`, но больше не вправе создавать объекты `Window`. Каждому производному классу придется реализовать метод `DrawWindow()`. Если производный класс не реализует абстрактный метод, он сам станет абстрактным, и создавать экземпляры этого класса будет невозможно.

Чтобы объявить метод абстрактным, следует в начале определения метода поставить ключевое слово `abstract`:

```
abstract public void DrawWindow();
```

(Поскольку у метода нет реализации, фигурные скобки отсутствуют, имеется только точка с запятой.)

Если один или несколько методов являются абстрактными, перед определением класса тоже должно стоять ключевое слово `abstract`:

```
abstract public class Window
```

В примере 5.3 демонстрируется создание абстрактного класса `Window` и абстрактного метода `DrawWindow()`.

Пример 5.3. Применение абстрактного метода и класса

```
using System;

abstract public class Window
{
   // конструктор принимает два целых числа -
   // координаты на экране
   public Window(int top, int left)
   {
      this.top = top;
      this.left = left;
   }
   // имитация рисования окна
   // обратите внимание: реализация отсутствует
   abstract public void DrawWindow();
```

```
   // эти элементы закрыты и потому невидимы
   // для методов производного класса,
   // что обсуждается далее в этой главе
   protected int top;
   protected int left;

}

// ListBox является наследником Window
public class ListBox : Window
{
   // в конструктор добавляется формальный параметр
   public ListBox(
      int top,
      int left,
      string contents):
      base(top, left)  // вызов конструктора базового класса
   {

      listBoxContents = contents;
   }

   // перегруженная версия, реализующая
   // абстрактный метод
   public override void DrawWindow()
   {

      Console.WriteLine ("Запись строки в список: {0}",
         listBoxContents);
   }

   private string listBoxContents;  // новая переменная класса
}

public class Button : Window
{
   public Button(
      int top,
      int left):
      base(top, left)
   {
   }

   // реализация абстрактного метода
   public override void DrawWindow()
   {
      Console.WriteLine("Рисуем кнопку с координатами {0}, {1}\n",
         top, left);
   }

}
public class Tester
{
   static void Main()
```

```
    {
        Window[] winArray = new Window[3];
        winArray[0] = new ListBox(1,2,"Первый список");
        winArray[1] = new ListBox(3,4,"Второй список");
        winArray[2] = new Button(5,6);

        for (int i = 0;i < 3; i++)
        {
            winArray[i].DrawWindow();
        }
    }
}
```

В примере 5.3 класс `Window` объявлен абстрактным, и, следовательно, для него нельзя создавать объекты. Если код первого элемента массива:

```
winArray[0] = new ListBox(1,2,"Первый список");
```

заменить на:

```
winArray[0] = new Window(1,2);
```

то компилятор сгенерирует следующее сообщение об ошибке:

```
Cannot create an instance of the abstract class or interface 'Window'
```

Зато вполне допустимо создавать объекты классов `ListBox` и `Button`, потому что эти классы переопределяют абстрактный метод, становясь от этого *конкретными* (то есть не абстрактными).

Ограничения абстрактных классов

Хотя объявление метода `DrawWindow()` абстрактным принуждает производные классы реализовать этот метод, такой подход все же страдает ограниченностью. Если произвести класс от класса `ListBox` (например `DropDownListBox`), то ничто не заставит производный класс реализовать собственный метод `DrawWindow()`.

Вниманию программистов, пишущих на языке C++! В C# метод `Window.DrawWindow()` не может предоставить реализацию, поэтому программисту не удастся в производных классах воспользоваться какими-нибудь процедурами, общими для всех методов `DrawWindow()`, которые могли бы быть реализованы в методе базового класса.

Наконец, абстрактные классы задуманы не просто как некий трюк реализации. Они представляют собой идею абстракции, которая является формой «договора» между всеми производными классами. Иными словами, абстрактные классы описывают открытые методы классов, реализующих абстракцию.

Идея абстрактного класса Window заключается в обрисовке общей схемы характеристик и поведения объектов типа Window, причем без намерения создавать экземпляры этого класса.

Сущность абстрактного класса заключается в слове «абстрактный». Он служит основой для реализации абстракции «окно», проявляющей себя в конкретных экземплярах окон, таких как окно броузера, кадр, кнопка, список, раскрывающийся список и т. д. Абстрактный класс формулирует, что такое Window, даже если у программиста нет намерения создавать «окно» как таковое. Альтернативой применению абстрактных классов является определение интерфейса, обсуждаемое в главе 8.

Изолированные классы

Полной противоположностью абстрактным классам являются *изолированные* (*sealed*) классы. В то время как абстрактный класс специально предназначен для создания производных классов и представляет собой их шаблон, изолированный класс вообще не может использоваться в качестве базового класса. Ключевое слово sealed, помещенное перед объявлением класса, запрещает создавать производные от него классы. Как правило, программисты помечают классы словом sealed, чтобы не допустить случайное наследование.

Если в объявлении класса Window в примере 5.3 заменить abstract на sealed (и убрать слово abstract из определения метода DrawWindow()), то программа не будет компилироваться. При попытке построить этот проект компилятор выдаст сообщение:

```
'ListBox' cannot inherit from sealed class 'Window'
```

а также много других сообщений (в том числе о невозможности создать защищенный (protected) элемент в изолированном классе).

Корень всех классов – класс Object

В C# считается, что все классы, независимо от их типа, в конечном счете произведены от класса System.Object. Интересно, что это относится и к размерным типам!

Базовый класс является непосредственным родителем производного класса. Производный класс сам может быть базовым для своих производных классов. Так появляется иерархическая структура – дерево наследования. Корневой класс – это самый верхний класс в такой иерархии. В языке C# корневым классом является класс Object. Если читателю подобная терминология не совсем понятна, он может представить себе перевернутое дерево, у которого корень вверху, а производные классы внизу. На таком дереве базовый класс расположен выше производного.

Класс Object обладает рядом методов, которые могут быть переопределены в производных классах (и действительно переопределяются в них). Среди таких методов можно назвать метод Equals(), определяющий, одинаковы ли два объекта; GetType(), возвращающий тип объекта (обсуждается в главе 18); а также ToString(), который возвращает строку, представляющую текущий объект (описан в главе 10). Методы класса Object перечислены в табл. 5.1.

Таблица 5.1. Методы класса Object

Метод	Описание
Equals()	Проверяет, эквивалентны ли два объекта
GetHashCode()	Позволяет объектам предоставлять собственные хеш-функции для использования в классах коллекций (см. главу 9)
GetType()	Предоставляет доступ к типу объекта (см. главу 18)
ToString()	Предоставляет строковое представление объекта
Finalize()	Освобождает ресурсы, не связанные с памятью, вызывается деструктором (см. главу 4)
MemberwiseClone()	Создает копии объекта; этот метод не следует реализовывать в своем типе
ReferenceEquals()	Проверяет, ссылаются ли два объекта на один и тот же экземпляр

Пример 5.4 демонстрирует применение метода ToString(), наследуемого от класса Object. В этом же коде показано, что ко встроенным типам данных (например int) можно относиться так, словно они наследуются от Object.

Пример 5.4. Наследование от класса Object

```
using System;

public class SomeClass
{
   public SomeClass(int val)
   {
      value = val;
   }

   public override string ToString()
   {
      return value.ToString();
   }
   private int value;
}
public class Tester
{
   static void Main()
   {
```

```
        int i = 5;
        Console.WriteLine("Значение i равно  {0}", i.ToString());

        SomeClass s = new SomeClass(7);
        Console.WriteLine("Значение s равно {0}", s.ToString());
    }
}
```

Вывод:

```
Значение i равно 5
Значение s равно 7
```

Документация метода `Object.ToString()` описывает его сигнатуру:

```
public virtual string ToString();
```

Это открытый виртуальный метод, который возвращает строку и не требует параметров. Все встроенные типы, такие как `int`, являются потомками класса `Object`, поэтому они могут вызывать его методы.

В примере 5.4 для класса `SomeClass` переопределяется виртуальная функция, что само по себе является обычной практикой. После переопределения метод `ToString()` будет возвращать некоторое осмысленное значение. Если закомментировать переопределенную функцию, то будет вызван метод базового класса, и тогда на экран будет выведено:

```
Значение s равно SomeClass
```

То есть по умолчанию этот метод возвращает строку с именем класса.

Нет нужды явно указывать, что классы произведены от `Object`, – это наследование подразумевается.

Упаковка и распаковка типов

Упаковка (*boxing*) и *распаковка* (*unboxing*) – это процессы, позволяющие трактовать размерные типы (например целочисленные) как ссылочные типы (объекты). Значение «упаковывается» в класс `Object`, а впоследствии «распаковывается» в исходный тип. Благодаря этому процессу в примере 5.4 удалось вызвать метод `ToString()` для переменной типа `int`.

Упаковка происходит неявно

Упаковка является неявным преобразованием размерного типа к типу `Object`. Упаковка какого-либо значения создает экземпляр класса `Object` и копирует это значение в созданный объект. Этот процесс проиллюстрирован на рис. 5.4.

Упаковка происходит неявно, когда размерный тип встречается в контексте, где ожидается ссылка. Например, если значение базового типа

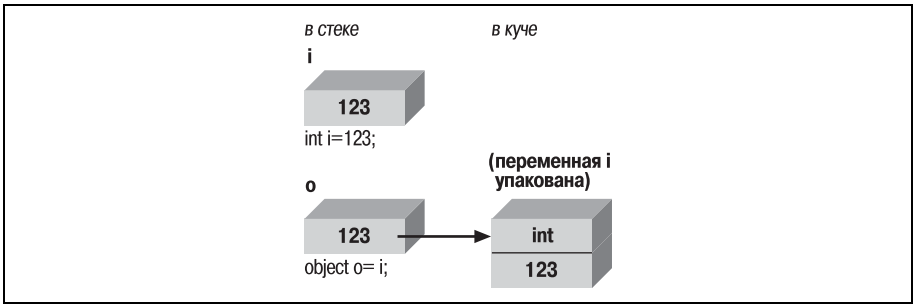

Рис. 5.4. *Упаковка типов*

(скажем, `int`) присвоить переменной типа `Object` (что вполне законно, так как `int` произведен от `Object`), то значение будет упаковано:

```
using System;
class Boxing
{   public static void Main()
    {
        int i = 123;
        Console.WriteLine("Значение объекта = {0}", i);
    }
}
```

Метод `Console.WriteLine()` ожидает объект, а отнюдь не целое число. Чтобы он работал, целый тип автоматически преобразуется библиотекой CLR, а для полученного объекта вызывается метод `ToString()`. Это свойство языка позволяет создавать методы, принимающие объект в качестве параметра. Такой метод будет работать независимо от того, передан ему ссылочный или размерный тип.

Распаковка должна быть явной

Чтобы преобразовать упакованный объект обратно в размерный тип, программист должен явно распаковать его. Точнее, необходимо выполнять следующие шаги:

1. Убедиться, что экземпляр объекта – не что иное, как упакованное значение данного типа.
2. Скопировать значение из экземпляра объекта в переменную соответствующего типа.

Распаковка показана на рис. 5.5.

Чтобы распаковка прошла успешно, обрабатываемый объект должен быть ссылкой на объект, созданный в процессе упаковки значения данного типа. Упаковка и распаковка иллюстрируются в примере 5.5.

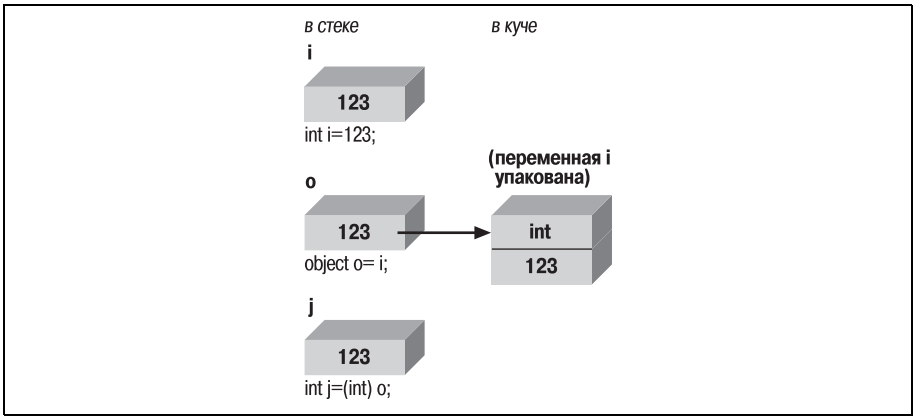

Рис. 5.5. *Упаковка и распаковка*

Пример 5.5. Упаковка и распаковка

```
using System;

public class UnboxingTest
{
   public static void Main()
   {
      int i = 123;

      // упаковка
      object o = i;

      // распаковка (должна быть явной)
      int j = (int) o;
      Console.WriteLine("j: {0}", j);
   }
}
```

В примере 5.5 создается целая переменная `i`, которая неявно упаковывается во время присваивания объекту `o`. Впоследствии это значение распаковывается и присваивается новой переменной типа `int`, значение которой выводится на экран.

Как правило, операция распаковки помещается в блок `try`, описанный в главе 11. Если распаковываемый объект равен `null` или является ссылкой на объект другого типа, вызывается исключение `InvalidCastException`.

Вложенные классы

Классы состоят из членов, и нет ничего невозможного в том, чтобы членом класса был тип, определенный пользователем. Так, класс `Button` может содержать поле типа `Location` (местоположение), а класс `Location` может содержать поле типа `Point` (точка). Наконец, класс `Point` может содержать поле типа `int`.

Иногда внутренний класс существует исключительно для обслуживания внешнего, и вовсе не требуется, чтобы он был виден где-то еще. (Короче говоря, внутренний класс действует как вспомогательный.) Вспомогательный класс можно определить в рамках определения внешнего. Внешний класс так и называется *внешним* (*outer*), а внутренний называют *вложенным* (*nested*).

Достоинством вложенных классов является то, что им доступны все элементы внешнего класса. Метод вложенного класса может обращаться ко всем закрытым переменным внешнего.

Кроме того, вложенный класс можно скрыть от всех других классов, определив его с модификатором `private`.

Наконец, открытый вложенный класс доступен во всей области видимости внешнего класса. Если класс `Outer` внешний, а открытый класс `Nested` вложен в него, то ссылаться на `Nested` следует как на `Outer.Nested`, причем внешний класс действует (в определенной степени) как пространство имен.

Вниманию программистов, пишущих на языке Java! Вложенные классы приблизительно эквивалентны статическим внутренним классам. В языке C# нет эквивалента нестатическим внутренним классам языка Java.

В примере 5.6 в класс `Fraction` помещается вложенный класс `FractionArtist`. Его задача – представить на экране простую дробь. В данном примере рисование на экране иммитируется двумя вызовами метода `WriteLine()`.

Пример 5.6. Применение вложенного класса

```
using System;
using System.Text;

public class Fraction
{
    public Fraction(int numerator, int denominator)
    {
        this.numerator=numerator;
        this.denominator=denominator;
    }

        // Методы опущены...

    public override string ToString()
    {
        StringBuilder s = new StringBuilder();
        s.AppendFormat("{0}/{1}",
            numerator, denominator);
        return s.ToString();
    }
```

```
    internal class FractionArtist
    {
        public void Draw(Fraction f)
        {
            Console.WriteLine("Рисуем числитель: {0}",
               f.numerator);
            Console.WriteLine("Рисуем знаменатель: {0}",
               f.denominator);
        }
    }
    private int numerator;
    private int denominator;
}

  public class Tester
{
  static void Main()
  {
        Fraction f1 = new Fraction(3,4);
        Console.WriteLine("f1: {0}", f1.ToString());

        Fraction.FractionArtist fa = new Fraction.FractionArtist();
        fa.Draw(f1);
  }
}
```

Вложенный класс выделен полужирным шрифтом. У класса `FractionArtist` только один элемент – метод `Draw()`. Особый интерес представляет тот факт, что `Draw()` имеет доступ к закрытым элементам `f.numerator` и `f.denominator`, которые не были бы видны ему, не будь он методом вложенного класса.

Обратите внимание, что для создания экземпляра вложенного класса в методе `Main()` приходится указывать имя внешнего класса:

```
Fraction.FractionArtist fa = new Fraction.FractionArtist();
```

Область видимости класса `FractionArtist` ограничена классом `Fraction`.

6

Перегрузка операций

Одна из целей языка C# состоит в том, чтобы обеспечить классам, определенным пользователем, обладание всей функциональностью базовых типов. Предположим, программист определяет тип, представляющий простые дроби. Обеспечить этот класс функциональностью базовых типов – значит, получить возможность выполнять арифметические операции над объектами класса (складывать две дроби, умножать их и т. д.), а также преобразовывать дроби в базовые типы (например, в `int`) и обратно. Конечно, программист, определивший тип, может реализовать методы для каждой из требуемых операций и вызывать их примерно так:

```
Fraction theSum = firstFraction.Add(secondFraction);
```

Это работает, но выглядит некрасиво, поскольку с базовыми типами обращаются не так. Было бы гораздо лучше написать:

```
Fraction theSum = firstFraction + secondFraction;
```

Такие операторы интуитивно понятны и написаны в том же стиле, что и операторы сложения базовых типов, например `int`.

В этой главе будет продемонстрирована технология определения стандартных операций для пользовательских типов. Кроме того, читатель узнает, как добавлять операции преобразования, чтобы пользовательские типы явно и неявно преобразовывались в другие типы.

Ключевое слово operator

В языке C# операции являются статическими методами, которые возвращают результат операции, а в качестве аргументов используют ее

операнды. Когда программист задает операцию для какого-либо класса, он перегружает соответствующий метод аналогично тому, как перегрузил бы любой метод класса, если бы это потребовалось. Так, чтобы перегрузить операцию сложения (+), следует написать:

```
public static Fraction operator+(Fraction lhs, Fraction rhs)
```

Здесь имена параметров, `lhs` и `rhs`, выбраны автором, а не продиктованы синтаксисом. Имя `lhs` обозначает операнд, стоящий слева от знака операции, а `rhs` – операнд, стоящий справа.

Синтаксис перегрузки операции в языке C# предписывает указывать ключевое слово `operator` и знак перегружаемой операции после него. Слово `operator` является модификатором метода. Поэтому, чтобы перегрузить операцию сложения (+), следует указать `operator+`.

Теперь, если написать:

```
Fraction theSum = firstFraction + secondFraction;
```

то будет вызвана перегруженная операция +, которой переменная `firstFraction` будет передана в качестве первого параметра, а переменная `secondFraction` – в качестве второго. Когда компилятор встречает выражение:

```
firstFraction + secondFraction
```

он преобразует его в:

```
Fraction.operator+(firstFraction, secondFraction)
```

В результате возвращается новое значение типа `Fraction`, которое в данном случае присваивается объекту `Fraction` с именем `theSum`.

Вниманию программистов, пишущих на языке C++!
В языке C# невозможно создавать нестатические операции, поэтому бинарные операции обязательно должны иметь два операнда.

Поддержка других языков платформы .NET

C# позволяет программисту перегружать операции для созданных им классов, хотя, строго говоря, это не соответствует спецификации CLS. Другие языки платформы .NET, например VB.NET, могут и не поддерживать перегрузку операций, поэтому необходимо, чтобы созданный класс предоставлял этим языкам альтернативные методы, дающие тот же эффект.

Например, когда программист перегружает операцию сложения (+), он поступит правильно, если создаст метод `add()`, делающий ту же работу.

Перегрузка операции должна давать лишь синтаксическое средство сокращения записи, а не единственно возможный способ, с помощью которого объекты решают поставленную задачу.

Создание новых операций

Перегрузка операций может сделать программу интуитивно понятнее, если она будет выглядеть как программа, обрабатывающая базовые типы. С другой стороны, она может превратить программу в нечто неуправляемое и бестолковое, если программист нарушит парадигму применения операций. Нельзя поддаваться искушению использовать операции нетрадиционно и в соответствии с личными вкусами.

Пусть, например, имеется класс, описывающей сотрудника фирмы. Может показаться заманчивым переопределить операцию инкремента (++) для этого класса так, чтобы вызывался метод, увеличивающий зарплату сотрудника. Однако такое переопределение лишь собьет с толку пользователей класса. Следует применять перегрузку операций осторожно и только тогда, когда ее смысл ясен и не противоречит обычному поведению базовых типов.

Логические пары

Общепринятой практикой является перегрузка операции проверки на равенство (==) для сравнения двух объектов (хотя равенство объектов может быть определено иначе). В языке C# требуется, чтобы, перегрузив операцию равенства, программист перегрузил и операцию неравенства (!=). Аналогичным образом спарены операции «больше» (>) и «меньше» (<), а также «больше или равно» (>=) и «меньше или равно» (<=).

Операция проверки на равенство

Если программист перегружает операцию равенства (==), рекомендуется, чтобы он перегрузил также и виртуальный метод `Equals()` класса `Object` так, чтобы действие метода соответствовало действию операции. Тогда класс с перегруженной операцией и методом будет полиморфным и совместимым с другими языками платформы .NET, которые не поддерживают перегрузку операторов (но разрешают перегружать методы). Классы FCL не пользуются перегруженными операциями, но они ожидают, что в классах, созданных пользователем, реализованы соответствующие методы. Например, класс `ArrayList` ожидает, что реализован метод `Equals()`.

Класс `Object` реализует метод `Equals()` со следующей сигнатурой:

```
public override bool Equals(object o)
```

Перегрузив этот метод, программист позволяет своему классу Fraction действовать полиморфно с другими объектами. В теле метода Equals() потребуется выполнить сравнение с другим объектом Fraction. В этом случае можно передать реализацию вместе с определением операции равенства:

```
public override bool Equals(object o)
{
    if (! (o is Fraction) )
    {
        return false;
    }
    return this == (Fraction) o;
}
```

Здесь операция is используется для проверки совместимости типа объекта с типом операнда (в данном случае с Fraction). Выражение o is Fraction истинно, если o имеет тип, совместимый с Fraction.

Операции преобразования типов

Язык C# неявным образом преобразует тип int в тип long и позволяет программисту явным образом выполнить обратное преобразование. Преобразование значения типа int в значение типа long производится *неявным образом*, поскольку заведомо известно, что любое значение типа int поместится в участке памяти, отведенном под тип long. Обратная операция преобразования значения типа long в значение типа int должна выполняться *явным образом* (с применением приведения типов), поскольку существует вероятность потери информации:

```
int myInt = 5;
long myLong;
myLong = myInt;         // неявное преобразование типов
myInt = (int) myLong;   // явное преобразование типов
```

Подобные функциональные возможности требуются и классу, созданному программистом, например классу Fraction. Для типа int допустимо использование неявного преобразования в дробь, поскольку любое целое число может быть представлено как дробь со знаменателем 1 (например, 15 == 15/1).

Для дроби можно разрешить явное преобразование в целое число, отдавая себе отчет в возможности потери данных. Например, 9/4 будет преобразовано в целое число 2.

Ключевое слово implicit применяется, когда преобразование типа гарантированно пройдет без потери информации. В противном случае следует указать ключевое слово explicit.

В примере 6.1 приводится возможный вариант реализации явного и неявного преобразования типов, а также некоторых операций над клас-

сом Fraction. (В примере несколько раз вызывается метод Console.WriteLine() для вывода сообщений о том, какому методу передается управление, однако гораздо лучшим решением было бы применение отладчика. Читатель может поместить точки останова в каждый тестируемый оператор и выполнять код в пошаговом режиме, следя за вызовом конструкторов.) При компиляции этого примера появится несколько предупреждений, поскольку метод GetHashCode() не реализован (см. главу 9).

Пример 6.1. Определение операций преобразования типа и некоторых других операций для класса Fraction

```
using System;

public class Fraction
{
    public Fraction(int numerator, int denominator)
    {
        Console.WriteLine("Конструктор Fraction (int, int)");
        this.numerator=numerator;
        this.denominator=denominator;
    }

    public Fraction(int wholeNumber)
    {
        Console.WriteLine("Конструктор Fraction (int)");
        numerator = wholeNumber;
        denominator = 1;
    }

    public static implicit operator Fraction(int theInt)
    {
        System.Console.WriteLine("Неявное преобразование в тип Fraction");
        return new Fraction(theInt);
    }

    public static explicit operator int(Fraction theFraction)
    {
        System.Console.WriteLine("Явное преобразование в тип int");
        return theFraction.numerator /
            theFraction.denominator;
    }

   public static bool operator==(Fraction lhs, Fraction rhs)
    {
        Console.WriteLine("Операция ==");
        if (lhs.denominator == rhs.denominator &&
            lhs.numerator == rhs.numerator)
        {
            return true;
        }
        // здесь должно быть реализовано приведение дробей
       return false;
   }
```

```
    public static bool operator !=(Fraction lhs, Fraction rhs)
    {
        Console.WriteLine("Операция !=");

        return !(lhs==rhs);
    }

    public override bool Equals(object o)
    {
        Console.WriteLine("Метод Equals");
        if (! (o is Fraction) )
        {
            return false;
        }
        return this == (Fraction) o;
    }

    public static Fraction operator+(Fraction lhs, Fraction rhs)
    {
        Console.WriteLine("Операция +");
        if (lhs.denominator == rhs.denominator)
        {
            return new Fraction(lhs.numerator+rhs.numerator,
                lhs.denominator);
        }

        // упрощенное решение для дробей с неравными знаменателями
        // 1/2 + 3/4 == (1*4 + 3*2) / (2*4) == 10/8
        int firstProduct = lhs.numerator * rhs.denominator;
        int secondProduct = rhs.numerator * lhs.denominator;
        return new Fraction(
            firstProduct + secondProduct,
            lhs.denominator * rhs.denominator
        );

    }

    public override string ToString()
    {
        String s = numerator.ToString() + "/" +
            denominator.ToString();
        return s;
    }

    private int numerator;
    private int denominator;
}
  public class Tester
{
  static void Main()
  {
        Fraction f1 = new Fraction(3,4);
        Console.WriteLine("f1: {0}", f1.ToString());
```

```
        Fraction f2 = new Fraction(2,4);
        Console.WriteLine("f2: {0}", f2.ToString());

        Fraction f3 = f1 + f2;
        Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString());

        Fraction f4 = f3 + 5;
        Console.WriteLine("f3 + 5 = f4: {0}", f4.ToString());

        Fraction f5 = new Fraction(2,4);
        if (f5 == f2)
        {
            Console.WriteLine("F5: {0} == F2: {1}",
                f5.ToString(),
                f2.ToString());
        }

    }
}
```

Класс `Fraction` начинается с определения двух конструкторов. Один из них принимает в качестве формальных параметров числитель и знаменатель, другой – целое число. Вслед за конструкторами объявляются две операции преобразования типов. Первая преобразует целое в тип `Fraction`:

```
public static implicit operator Fraction(int theInt)
{
   return new Fraction(theInt,1);
}
```

Это преобразование помечено как `implicit` (неявное), поскольку любое целое число (`int`) легко преобразуется в тип `Fraction`: числителю присваивается это число, а знаменатель устанавливается в `1`. Ответственность за такое преобразование делегируется конструктору, принимающему в качестве параметров числитель (данное целое число) и знаменатель (1).

Вторая операция преобразования предназначена для явного перевода дробей в целые числа:

```
public static explicit operator int(Fraction theFraction)
{
   return theFraction.numerator /
        theFraction.denominator;
}
```

Поскольку здесь задействовано целочисленное деление, результат получается усеченным. То есть дробь `16/15` будет преобразована в `1`. В принципе, можно реализовать более тонкую операцию преобразования, где выполнялось бы округление.

После операций преобразования определены операции проверки равенства (`==`) и неравенства (`!=`). Читатель, конечно, помнит, что реализация одной из них влечет за собой обязательную реализацию другой.

Равенство для типа Fraction сформулировано как одновременное равенство числителей и знаменателей. Таким образом, дроби 3/4 и 6/8 не будут считаться равными. И здесь возможна более тонкая реализация, такая, где дроби сокращались бы перед сравнением.

Далее в примере переопределяется метод Equals() класса Object, чтобы объекты типа Fraction трактовались полиморфно с любым другим объектом. Данная реализация делегирует определение равенства оператору эквивалентности.

Безусловно, в классе Fraction должны быть реализованы все арифметические операции (сложение, вычитание, умножение, деление). Чтобы не загромождать пример, автор предлагает реализацию только сложения, да и она упрощена до предела. Знаменатели сравниваются друг с другом, и если они равны, числители складываются:

```
public static Fraction operator+(Fraction lhs, Fraction rhs)
{
    if (lhs.denominator == rhs.denominator)
    {
        return new Fraction(lhs.numerator+rhs.numerator,
            lhs.denominator);
    }
```

Если знаменатели не равны, выполняется перекрестное умножение:

```
int firstProduct = lhs.numerator * rhs.denominator;
int secondProduct = rhs.numerator * lhs.denominator;
return new Fraction(
    firstProduct + secondProduct,
    lhs.denominator * rhs.denominator
```

Этот код лучше объяснить на примере. При сложении 1/2 и 3/4 первый числитель (1) умножается на второй знаменатель (4), а результат сохраняется в переменной firstProduct. Второй числитель (3) умножается на первый знаменатель (2), и результат записывается в secondProduct. Сложение этих двух результатов (6+4) дает числитель (10) окончательного ответа. Чтобы получить знаменатель, нужно перемножить два исходных знаменателя (2×4=8). Полученная дробь (10/8) и является ответом.[1]

Наконец, чтобы облегчить отладку нового класса Fraction, программа написана так, что класс возвращает значение в виде строки, имеющей формат «числитель/знаменатель»:

```
public override string ToString()
{
   String s = numerator.ToString() + "/" +
      denominator.ToString();
```

[1] Повторим: 1/2=4/8, 3/4=6/8, 4/8+6/8=10/8. Для простоты сокращение дроби не производится.

```
    return s;
}
```

Новый строковый объект создается вызовом метода ToString() для числителя. Поскольку numerator имеет тип int, а это размерный тип, то вызов ToString() заставляет компилятор неявно упаковать целое (создавая при этом объект) и вызвать метод ToString() этого объекта, который, в свою очередь, возвращает строковое представление числителя. Эта строка объединяется со строкой «/», а результат – со строкой, возвращенной методом ToString(), вызванным для знаменателя.

Теперь класс Fraction можно протестировать. Сначала создадим простые дроби, 3/4 и 2/4:

```
Fraction f1 = new Fraction(3,4);
Console.WriteLine("f1: {0}", f1.ToString());

Fraction f2 = new Fraction(2,4);
Console.WriteLine("f2: {0}", f2.ToString());
```

Результат работы этого кода нетрудно предвидеть – выводится запись о вызове конструктора, а затем – значения:

```
Конструктор Fraction (int, int)
f1: 3/4
Конструктор Fraction (int, int)
f2: 2/4
```

Следующая строчка метода Main() вызывает статическую операцию «+». Предназначение этой операции – сложить две дроби и вернуть результат в виде дроби:

```
Fraction f3 = f1 + f2;
Console.WriteLine("f1 + f2 = f3: {0}", f3.ToString());
```

Изучая результат работы программы, легко понять, как работает операция «+»:

```
Операция +
Конструктор Fraction (int, int)
f1 + f2 = f3: 5/4
```

После вызова операции «+» вызывается конструктор для объекта f3. Конструктор принимает два целых значения, числитель и знаменатель, а возвращает новую дробь.

Далее в методе Main() к переменной f3 типа Fraction прибавляется целое значение, а результат заносится в новую переменную типа Fraction, а именно f4:

```
Fraction f4 = f3 + 5;
Console.WriteLine("f3 + 5: {0}", f4.ToString());
```

В создаваемом программой тексте раскрываются все производимые преобразования:

```
Неявное преобразование в тип Fraction
Конструктор Fraction (int)
Операция +
Конструктор Fraction (int, int)
f3 + 5 =  f4: 25/4
```

Обратите внимание, что значение 5 было неявно преобразовано в дробь. В операторе `return` метода, выполняющего неявное преобразование, вызывается конструктор `Fraction`, создающий дробь `5/1`. Эта дробь вместе с дробью `f3` передается операции сложения, а сумма передается конструктору объекта `f4`.

Наконец, в методе `Main()` создается новая дробь (`f5`), и выполняется проверка, равна ли она дроби `f2`. Если это так, оба значения выводятся:

```
Fraction f5 = new Fraction(2,4);
if (f5 == f2)
{
    Console.WriteLine("F5: {0} == F2: {1}",
        f5.ToString(),
        f2.ToString());
}
```

Выводимый текст отражает факт создания дроби `f5` и последующий вызов перегруженной операции проверки на равенство:

```
Конструктор Fraction (int, int)
Операция ==
F5: 2/4 == F2: 2/4
```

7

Структуры

Структура (*struct*) – это простой определяемый пользователем тип, являющийся облегченной альтернативой классам. Структуры аналогичны классам в том смысле, что могут иметь конструкторы, свойства, методы, поля, операторы, вложенные типы и индексаторы (см. главу 9).

Между структурами и классами существует и много различий. Например, структуры не поддерживают наследование и не имеют деструкторов. Более важное различие состоит в том, что классы относятся к ссылочным типам ссылок, а структуры – к размерным типам. (Более подробная информация о классах и типах приведена в главе 3.) Таким образом, структуры годятся для представления объектов, к которым не предполагается обращаться по ссылке.

Общепринятая точка зрения состоит в том, что структурами следует пользоваться только для типов, которые невелики по размеру, просты и по своему поведению аналогичны встроенным.

Структуры несколько более эффективно используют память, если они реализованы в массивах (см. главу 9). В классах коллекций они расходуют память не так экономно. Классы коллекций используют ссылки, а для создания ссылок на структуры они должны быть упакованы. Во избежание накладных расходов на упаковку и распаковку в больших объектах коллекций предпочтительнее пользоваться классами.

В этой главе будет показано, как определять структуры и работать с ними, а также как инициализировать их значения с помощью конструкторов.

Определение структур

Синтаксис определения структуры почти такой же, как у определения класса:

```
[атрибуты] [модификаторы-доступа] struct [:список интерфейса]
{элементы-структуры}
```

В примере 7.1 иллюстрируется определение структуры. Структура `Location` представляет собой точку на двумерной поверхности. Обратите внимание, что эта структура объявляется в точности как класс, если не считать присутствия ключевого слова `struct`. Кроме того, обратите внимание, что конструктор структуры `Location` принимает два целых значения и присваивает их элементам экземпляра, `x` и `y`. Координаты `x` и `y` структуры `Location` объявляются как свойства.

Пример 7.1. Создание структуры

```
using System;

public struct Location
{
    public Location(int xCoordinate, int yCoordinate)
    {
        xVal = xCoordinate;
        yVal = yCoordinate;
    }

    public int x
    {
        get
        {
            return xVal;
        }
        set
        {
            xVal = value;
        }
    }

    public int y
    {
        get
        {
            return yVal;
        }
        set
        {
            yVal = value;
        }
    }

    public override string ToString()
    {
```

```
        return (String.Format("{0}, {1}", xVal,yVal));
    }

    private int xVal;
    private int yVal;
}

public class Tester
{
    public void myFunc(Location loc)
    {
        loc.x = 50;
        loc.y = 100;
        Console.WriteLine("В MyFunc координаты Loc1: {0}", loc);
    }
    static void Main()
    {
        Location loc1 = new Location(200,300);
        Console.WriteLine("Координаты Loc1: {0}", loc1);
        Tester t = new Tester();
        t.myFunc(loc1);
        Console.WriteLine("Координаты Loc1: {0}", loc1);
    }
}
```

Вывод:

```
Координаты Loc1: 200, 300
В MyFunc координаты Loc1: 50, 100
Координаты Loc1: 200, 300
```

В отличие от классов, структуры не поддерживают наследование. Они являются неявными потомками класса `Object` (как и все типы C#, включая базовые), но не могут наследовать ни от какого другого класса или структуры. Структуры являются неявно *изолированными* (то есть никакой класс или структура не могут быть произведены от структуры). Однако подобно классам структуры могут реализовывать любое число интерфейсов. Дополнительные отличия структур от классов заключаются в следующем:

Отсутствие деструктора или пользовательского конструктора, используемого по умолчанию

У структур не может быть деструкторов, а также пользовательских конструкторов без параметров (вызываемых по умолчанию). Если вообще не указать никакого конструктора, то структуру создаст конструктор по умолчанию, который обнулит все элементы данных или установит их в значения, подходящие к их типу (см. табл. 4.2). Если программист определяет конструктор, он должен инициализировать поля структуры.

Отсутствие инициализации

Поле объекта структуры инициализировать нельзя. Иными словами, нельзя написать:

```
private int xVal = 50;
private int yVal = 100;
```

хотя в отношении полей класса это вполне допустимо.

Структуры задумывались как простые и облегченные конструкции языка. В то время как закрытые элементы способствуют сокрытию данных и инкапсуляции, некоторые программисты полагают, что для структур это излишне. Они объявляют данные открытыми, чем упрощают реализацию структуры. Другие программисты считают, что свойства обеспечивают ясный и простой интерфейс и что хороший стиль программирования требует сокрытия данных даже в случае простых облегченных объектов. Какую из двух стратегий выбрать – вопрос из области философии разработки программного обеспечения. Язык C# поддерживает оба подхода.

Создание структур

Объект структуры создается с помощью ключевого слова `new` в операторе присваивания аналогично тому, как это делается для класса. В примере 7.1 класс `Tester` создает экземпляр структуры `Location`:

```
Location loc1 = new Location(200,300);
```

Здесь новый объект называется `loc1`, а конструктору передаются два значения, `200` и `300`.

Структуры как размерные типы

Определение класса `Tester` в примере 7.1 включает в себя объект `loc1` типа `Location`, созданный со значениями `200` и `300`. Конструктор вызывается так:

```
Location loc1 = new Location(200,300);
```

Затем вызывается метод `WriteLine()`:

```
Console.WriteLine("Loc1 location: {0}", loc1);
```

В аргументе этого метода должен передаваться объект, но тип `Location` является структурой (то есть имеет размерный тип). Компилятор автоматически упаковывает структуру (как упаковал бы любой другой размерный тип), и методу `WriteLine()` передается объект – результат упаковки. Для него вызывается метод `ToString()`, и, поскольку структура (неявно) наследует от класса `Object`, она в состоянии отреагировать полиморфно. Иными словами, она переопределяет метод, как поступил бы любой другой объект в подобной ситуации:

```
Координаты Loc1: 200, 300
```

Однако структуры являются объектами, имеющими размерный тип. Поэтому они передаются функции по значению, что видно из следующей строчки кода, где объект loc1 передается методу myFunc():

```
t.myFunc(loc1);
```

Внутри этого метода элементам x и y присваиваются значения, которые затем выводятся на консоль:

```
В MyFunc координаты Loc1: 50, 100
```

После возврата управления вызвавшему методу (то есть методу Main()) снова вызывается WriteLine(). Значения остаются прежними:

```
Координаты Loc1: 200, 300
```

Структура была передана функции как объект, имеющий размерный тип, и в теле функции была создана копия структуры. Поставим эксперимент и объявим Location как класс:

```
public class Location
```

Выполним программу снова. Будет выведена следующая информация:

```
Координаты Loc1: 200, 300
В MyFunc координаты Loc1: 50, 100
Координаты Loc1: 50, 100
```

На этот раз объект Location несет в себе ссылочную семантику. Так, при изменении значений в функции myFunc() они изменяются и в самом объекте, созданном в Main().

Вызов конструктора, используемого по умолчанию

Как было сказано выше, если программист не напишет конструктор, компилятор вызовет используемый по умолчанию. В этом можно убедиться, если закомментировать определение конструктора:

```
/* public Location(int xCoordinate, int yCoordinate)
    {
        xVal = xCoordinate;
        yVal = yCoordinate;
    }
*/
```

а также заменить первую строчку метода Main() на такую, где экземпляр Location создается без передачи значений:

```
// Location loc1 = new Location(200,300);
 Location loc1 = new Location();
```

Поскольку в программе отсутствует конструктор, вызывается конструктор, используемый по умолчанию. В результате ее выполнения выводится следующий текст:

```
Координаты Loc1: 0, 0
В MyFunc координаты Loc1: 50, 100
Координаты Loc1: 0, 0
```

Конструктор, используемый по умолчанию, проинициализировал поля структуры нулями.

Вниманию программистов, пишущих на языке C++! В C# ключевое слово new не всегда создает объекты в куче. Классы размещаются в куче, а структуры – в стеке. Кроме того, если ключевое слово new опущено (как будет показано в следующем разделе), конструктор вообще не вызывается. Из-за того что в C# требуется явная инициализация, программист должен явно инициализировать все члены структуры до ее использования.

Создание структур без ключевого слова new

Поскольку loc1 является структурой (а не классом), она создается в стеке. Так, в примере 7.1, когда выполняется оператор new:

```
Location loc1 = new Location(200,300);
```

объект типа Location будет создан в стеке.

Оператор new вызывает конструктор Location. Однако, в отличие от класса, структуру можно создать и без помощи оператора new. Это соответствует общим принципам создания переменных базового типа (например int). Иллюстрация сказанного дана в примере 7.2.

Предостережение: автор демонстрирует здесь, как создавать структуры без ключевого слова new, поскольку в этом проявляется отличие C# от C++ и отличие классов от структур в самом C#. В то же время, создание структур без ключевого слова new не дает ощутимой выгоды, зато усложняет чтение и сопровождение программ и делает их менее защищенными от ошибок! Программист должен помнить, что действует на свой страх и риск.

Пример 7.2. Создание структур без ключевого слова new

```
using System;

public struct Location
{
    public Location(int xCoordinate, int yCoordinate)
    {
        xVal = xCoordinate;
        yVal = yCoordinate;
    }
      public int x
    {
```

```
        get
        {
            return xVal;
        }
        set
        {
            xVal = value;
        }
    }

    public int y
    {
        get
        {
            return yVal;
        }
        set
        {
            yVal = value;
        }
    }

    public override string ToString()
    {
        return (String.Format("{0}, {1}", xVal,yVal));
    }

    public int xVal;
    public int yVal;
}

public class Tester
{
    static void Main()
    {
        Location loc1;          // вызов конструктора отсутствует
        loc1.xVal = 75;         // инициализация элементов
        loc1.yVal = 225;
        Console.WriteLine(loc1);
    }
}
```

В примере 7.2 локальные переменные инициализируются непосредственно, до вызова метода `WriteLine()` и передачи ему объекта `loc1` в качестве аргумента:

```
loc1.xVal = 75;
loc1.yVal = 225;
```

Если закомментировать одну из операций присваивания и перекомпилировать программу:

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

```
    Location loc1;
    loc1.xVal = 75;
 //   loc1.yVal = 225;
    Console.WriteLine(loc1);
}
```

то компилятор выдаст ошибку:

```
Use of unassigned local variable 'loc1'
```

После присваивания всех значений они становятся доступными через свойства `x` и `y`:

```
static void Main()
{
    Location loc1;
    loc1.xVal = 75;              // присваивание значения элементу
    loc1.yVal = 225;             // присваивание значения элементу
    loc1.x = 300;                // использование свойства
    loc1.y = 400;                // использование свойства
    Console.WriteLine(loc1);
}
```

Свойствами следует пользоваться осторожно. Хотя они и позволяют поддерживать инкапсуляцию, делая значения закрытыми, сами свойства фактически являются методами класса, а метод нельзя вызвать, пока не выполнена инициализация всех переменных.

8

Интерфейсы

Интерфейс – это контракт, обеспечивающий определенное поведение класса или структуры. Когда класс реализует интерфейс, он как бы говорит потенциальному пользователю: «Я гарантирую, что поддерживаю методы, свойства, события и индексаторы этого интерфейса». (Более подробную информацию о методах и свойствах см. в главе 4, о событиях – в главе 12, об индексаторах – в главе 9.)

Интерфейс представляет собой альтернативу абстрактному классу в смысле создания контрактов между классами и их пользователями. Эти контракты создаются с использованием ключевого слова `interface`, которое объявляет ссылочный тип, инкапсулирующий контракт.

Синтаксически интерфейс подобен классу, имеющему только абстрактные методы. Абстрактный класс является базовым для семейства производных классов, в то время как интерфейсы задуманы для «смешивания» их с прочими деревьями наследования.

Когда класс реализует интерфейс, он должен реализовать все методы этого интерфейса. По сути дела, он соглашается выполнить контракт, определенный интерфейсом.

Наследование от абстрактного класса реализует отношение *является*, рассмотренное в главе 5. Реализация интерфейса определяет другое отношение, до сих пор еще не обсуждавшееся, – *реализует*. Между этими двумя отношениями существует тонкая разница. Автомобиль *является* транспортным средством, но он может *реализовать* функциональную возможность `CanBeBoughtWithABigLoan` (ПродаетсяСБольшойСсудой) аналогично, например, продаже дома.

В этой главе описывается, как создавать, реализовывать и применять интерфейсы. Читатель узнает, как реализовать множественные интер-

фейсы и как комбинировать и расширять их, а также как проверять, реализовал ли класс какой-либо интерфейс.

Подмешивание

В Сомервилле, штат Массачусетс, одно время было кафе-мороженое, где леденцы и другие лакомства подмешивались к различным сортам мороженого. В те же годы в расположенном неподалеку Массачусетском технологическом институте группа пионеров объектно-ориентированного подхода к программированию разрабатывала язык программирования SCOOPS. Похоже, случайное совпадение этого названия с английским словом «scoop» (ковш, черпак) вызвало у этих ребят какие-то ассоциации, и они ввели термин «подмешивание» (mix in) для классов, подмешивающих дополнительные функциональные возможности. Эти «подмешивающие» классы играли практически ту же роль, что и интерфейсы C#.

Реализация интерфейса

Синтаксис определения интерфейса следующий:

```
[атрибуты] [модификатор-доступа] interface имя-интерфейса [:список-базовых-интерфейсов] {тело-интерфейса}
```

Пусть читатель пока не обращает внимание на атрибуты – они обсуждаются в главе 18.

Модификаторы права доступа, а именно `public`, `private`, `protected`, `internal` и `protected internal`, были подробно рассмотрены в главе 4.

После ключевого слова `interface` стоит имя интерфейса. Принято (но не требуется) начинать имя интерфейса с заглавной буквы I, например `IStorable`, `ICloneable`, `IClaudius` и т. д.

В *списке базовых интерфейсов* перечислены интерфейсы, расширяемые данным (подробности в разделе «Реализация нескольких интерфейсов» далее в этой главе).

Тело интерфейса – это его реализация, описываемая ниже. Предположим, что нужно создать интерфейс, описывающий методы и свойства, необходимые классу для того, чтобы его можно было сохранять в специально отведенном месте (в базе данных или файле) и читать оттуда. Назовем этот интерфейс `IStorable`.

Интерфейсу потребуются два метода, `Read()` и `Write()`, которые и появятся в его теле:

```
interface IStorable
{
```

```
   void Read();
   void Write(object);
}
```

Цель интерфейса – определить функциональные возможности, которыми должен обладать класс.

Например, программист создает класс Document. Оказывается, что объекты типа Document можно сохранять в базе данных, и тогда программист принимает решение, что класс Document должен реализовать интерфейс IStorable.

При реализации интерфейса требуется придерживаться того же синтаксиса, что и при наследовании. А именно ставится двоеточие (:), за которым указывается имя интерфейса:

```
public class Document : IStorable
{
   public void Read() {...}
   public void Write(object obj) {...}
   // ...
}
```

Теперь автор класса Document несет ответственность за осмысленную реализацию методов интерфейса IStorable. Объявив, что класс Document реализует интерфейс IStorable, программист должен реализовать все методы этого интерфейса, иначе компилятор выдаст сообщение об ошибке. Сказанное иллюстрируется примером 8.1, в котором класс Document реализует интерфейс IStorable.

Пример 8.1. Применение простого интерфейса

```
using System;

// объявление интерфейса
interface IStorable
{
   // модификаторы доступа отсутствуют, методы открыты, реализация отсутствует
   void Read();
   void Write(object obj);
   int Status { get; set; }

}

// создание класса, реализующего интерфейс IStorable
public class Document : IStorable
{
   public Document(string s)
   {
      Console.WriteLine("Создание документа: {0}", s);
   }

   // реализация метода Read
   public void Read()
   {
```

```
            Console.WriteLine(
                "Реализация метода Read для IStorable");
        }

        // реализация метода Write
        public void Write(object o)
        {
            Console.WriteLine(
                "Реализация метода Write для IStorable");
        }
        // реализация свойства
        public int Status
        {
            get
            {
                return status;
            }

            set
            {
                status = value;
            }
        }

        // сохранение значения свойства
        private int status = 0;
    }

    // проверка работы интерфейса
    public class Tester
    {

        static void Main()
        {
            // обращение к методам объекта Document
            Document doc = new Document("Test Document");
            doc.Status = -1;
            doc.Read();
            Console.WriteLine("Статус документа: {0}", doc.Status);

            // преобразование в тип интерфейса и применение интерфейса
            IStorable isDoc = (IStorable) doc;
            isDoc.Status = 0;
            isDoc.Read();
            Console.WriteLine("Статус IStorable: {0}", isDoc.Status);

        }
    }
```

Вывод:

```
Создание документа: Test Document
Реализация метода Read для IStorable
Статус документа: -1
Реализация метода Read для IStorable
Статус IStorable: 0
```

В примере 8.1 определяется простой интерфейс IStorable, имеющий два метода, Read() и Write(), и свойство Status целочисленного типа. Обратите внимание, что в объявлении свойства нет реализации методов get() и set(), а просто декларируется их наличие:

```
int Status { get; set; }
```

Обратите также внимание на тот факт, что в объявлении методов интерфейса IStorable нет модификаторов доступа (то есть ключевых слов public, protected, internal, private). Более того, наличие такого модификатора было бы воспринято компилятором как ошибка. Методы интерфейса открыты по умолчанию, поскольку интерфейс по своей сути является контрактом, которому следуют другие классы. Нельзя создать экземпляр интерфейса, зато можно создать экземпляр класса, реализующего интерфейс.

Класс, реализующий интерфейс, должен выполнить условие контракта точно и полно. Класс Document должен предоставить как методы Read() и Write(), так и свойство Status. *Как* он выполнит требования – это его дело. Хотя интерфейс IStorable предписывает классу Document иметь свойство Status, его не заботит, будет ли класс хранить соответствующее значение в переменной или в базе данных. Детали оставлены на усмотрение класса, реализующего интерфейс.

Реализация нескольких интерфейсов

Класс может реализовать и несколько интерфейсов. Например, если класс Document может быть сохранен и вдобавок сжат, ничто не мешает программисту реализовать два интерфейса – IStorable и ICompressible. Для этого в объявлении класса (в списке базовых интерфейсов) должны быть через запятую перечислены все реализуемые интерфейсы:

```
public class Document : IStorable, ICompressible
```

Теперь класс Document обязан реализовать и методы, указанные в интерфейсе ICompressible (см. пример 8.2):

```
public void Compress()
 {
     Console.WriteLine("Реализация метода Compress");
 }

 public void Decompress()
 {
     Console.WriteLine("Реализация метода Decompress");
 }
```

Расширение интерфейсов

Существующий интерфейс можно расширить, чтобы добавить новые методы или члены, либо модифицировать работу существующих эле-

ментов. Например, от интерфейса ICompressible можно произвести новый, ILoggedCompressible, который будет расширять исходный интерфейс методом, отслеживающим количество сохраненных байтов:

```
interface ILoggedCompressible : ICompressible
{
    void LogSavedBytes();
}
```

Теперь классы могут реализовать либо интерфейс ICompressible, либо ILoggedCompressible в зависимости от того, требуется ли им дополнительная функциональность. Если класс реализует ILoggedCompressible, он должен реализовать методы как интерфейса ILoggedCompressible, так и интерфейса ICompressible. Объекты этого класса будут приводиться либо к типу ILoggedCompressible, либо к типу ICompressible.

Комбинирование интерфейсов

Аналогичным образом можно создавать новые интерфейсы, сочетая уже существующие и, возможно, добавляя новые методы или свойства. Например, можно создать интерфейс IStorableCompressible, сочетающий методы обоих интерфейсов, который дополнительно имел бы метод, сохраняющий первоначальную длину сжатых данных:

```
interface IStorableCompressible : IStoreable, ILoggedCompressible
{
     void LogOriginalSize();
}
```

В примере 8.2 демонстрируется расширение и комбинирование интерфейсов.

Пример 8.2. Расширение и комбинирование интерфейсов

```
using System;

interface IStorable
{
   void Read();
   void Write(object obj);
   int Status { get; set; }

}

// новый интерфейс
interface ICompressible
{
   void Compress();
   void Decompress();
}

// расширение интерфейса
interface ILoggedCompressible : ICompressible
{
```

```
   void LogSavedBytes();
}

// комбинирование интерфейсов
interface IStorableCompressible : IStorable, ILoggedCompressible
{
   void LogOriginalSize();
}

// еще один интерфейс
interface IEncryptable
{
   void Encrypt();
   void Decrypt();
}

public class Document : IStorableCompressible, IEncryptable
{
   // конструктор класса Document
   public Document(string s)
   {
      Console.WriteLine("Создание документа: {0}", s);
   }

   // реализация интерфейса IStorable
   public void Read()
   {
      Console.WriteLine(
         "Реализация метода Read для IStorable");
   }

   public void Write(object o)
   {
      Console.WriteLine(
         "Реализация метода Write для IStorable");
   }

   public int Status
   {
      get
      {
         return status;
      }
      set
      {
         status = value;
      }
   }

   // реализация интерфейса ICompressible
   public void Compress()
   {
      Console.WriteLine("Реализация Compress");
   }
```

```
    public void Decompress()
    {
       Console.WriteLine("Реализация Decompress");
    }

    // реализация интерфейса ILoggedCompressible
    public void LogSavedBytes()
    {
       Console.WriteLine("Реализация LogSavedBytes");
    }

    // реализация интерфейса IStorableCompressible
    public void LogOriginalSize()
    {
       Console.WriteLine("Реализация LogOriginalSize");
    }

    // реализация интерфейса IEncryptable
    public void Encrypt()
    {
       Console.WriteLine("Реализация Encrypt");
    }

    public void Decrypt()
    {
       Console.WriteLine("Реализация Decrypt");
    }

    // переменная для хранения свойства Status интерфейса IStorable
    private int status = 0;
}

public class Tester
{

    static void Main()
    {
       // создание объекта Document
       Document doc = new Document("Test Document");

       // преобразование объекта Document в различные интерфейсы
       IStorable isDoc = doc as IStorable;
       if (isDoc != null)
       {
          isDoc.Read();
       }
       else
          Console.WriteLine("IStorable не поддерживается");

       ICompressible icDoc = doc as ICompressible;
       if (icDoc != null)
       {
          icDoc.Compress();
       }
```

```
        else
           Console.WriteLine("Compressible не поддерживается");

        ILoggedCompressible ilcDoc = doc as ILoggedCompressible;
        if (ilcDoc != null)
        {
           ilcDoc.LogSavedBytes();
           ilcDoc.Compress();
           // ilcDoc.Read();
        }
        else
           Console.WriteLine("LoggedCompressible не поддерживается");

        IStorableCompressible isc = doc as IStorableCompressible;
        if (isc != null)
        {
           isc.LogOriginalSize();  // IStorableCompressible
           isc.LogSavedBytes();    // ILoggedCompressible
           isc.Compress();         // ICompressible
           isc.Read();             // IStorable

        }
        else
        {
           Console.WriteLine("StorableCompressible не поддерживается");
        }

        IEncryptable ie = doc as IEncryptable;
        if (ie != null)
        {
           ie.Encrypt();
        }
        else
           Console.WriteLine("Encryptable не поддерживается");
     }
  }
```

Вывод:

```
Создание документа: Test Document
Реализация метода Read для IStorable
Реализация Compress
Реализация LogSavedBytes
Реализация Compress
Реализация LogOriginalSize
Реализация LogSavedBytes
Реализация Compress
Реализация метода Read Method для IStorable
Реализация Encrypt
```

Пример 8.2 начинается с реализации интерфейсов `IStorable` **и** `ICompressible`**. Последний расширяется интерфейсом** `ILoggedCompressible`**, а затем два интерфейса комбинируются в интерфейс** `IStorableCompressible`**. Наконец, добавляется новый интерфейс** `IEncryptable`**.**

Программа Tester создает новый объект Document и затем выполняет приведение типа для различных интерфейсов. Когда объект приведен к типу интерфейса ILoggedCompressible, этот интерфейс можно использовать для вызова методов интерфейса ICompressible, поскольку ILoggedCompressible расширяет (и тем самым делает своими) методы базового интерфейса:

```
ILoggedCompressible ilcDoc = doc as ILoggedCompressible;
if (ilcDoc != null)
{
    ilcDoc.LogSavedBytes();
    ilcDoc.Compress();
    // ilcDoc.Read();
}
```

Однако вызвать метод Read() нельзя, потому что это метод интерфейса IStorable, не связанного с данным. Если раскомментировать вызов метода Read(), компилятор выдаст сообщение об ошибке.

Когда объект приведен к типу IStorableCompressible (к типу интерфейса, в котором скомбинированы ILoggedCompressible и IStorable), становятся доступными методы всех трех интерфейсов: IStorableCompressible, ICompressible и IStorable:

```
IStorableCompressible isc = doc as IStorableCompressible
if (isc != null)
{
    isc.LogOriginalSize();  // IStorableCompressible
    isc.LogSavedBytes();    // ILoggedCompressible
    isc.Compress();         // ICompressible
    isc.Read();             // IStorable
}
```

Обращение к методам интерфейса

Программист может обращаться к членам интерфейса IStorable, как если бы они были членами класса Document:

```
Document doc = new Document("Test Document");
doc.status = -1;
doc.Read();
```

В качестве альтернативы можно создать объект интерфейса, выполнив операцию приведения типа объекта к типу этого интерфейса. После этого можно свободно использовать этот интерфейс для доступа к его методам:

```
IStorable isDoc = (IStorable) doc;
isDoc.status = 0;
isDoc.Read();
```

В данном случае программист *знает*, что в `Main()` объект `Document` на самом деле имеет тип `IStorable`, и может воспользоваться этим знанием.

Как было сказано выше, нельзя создать объект интерфейса непосредственно. То есть нельзя написать:

```
IStorable isDoc = new IStorable();
```

Зато допускается создание объекта класса, реализующего интерфейс:

```
Document doc = new Document("Test Document");
```

После чего можно создавать объект интерфейса, выполнив операцию приведения типа объекта реализующего класса к типу этого интерфейса, в данном случае – к `IStorable`:

```
IStorable isDoc = (IStorable) doc;
```

Описанные шаги можно слить в один:

```
IStorable isDoc = (IStorable) new Document("Test Document");
```

В общем случае более удачным решением будет обращение к методам интерфейса с использованием ссылки на интерфейс. Так, в рассмотренном выше примере лучше использовать запись `isDoc.Read()`, чем `doc.Read()`. Доступ через интерфейс позволяет трактовать интерфейс полиморфно. Другими словами, можно заставить два (и более) класса реализовать интерфейс, а затем обращаться к ним через этот интерфейс, игнорируя их реальный тип и не проводя между ними различия. Более подробно полиморфизм обсуждался в главе 5.

Приведение к типу интерфейса

Во многих случаях программист заранее не знает, поддерживает ли объект тот или иной интерфейс. Имея коллекцию объектов, программист не может сказать, поддерживает ли конкретный объект интерфейс `IStorable` или `ICompressible`, или оба интерфейса. Зато он *может* выполнить операцию приведения типа объектов к типу этих интерфейсов:

```
Document doc = new Document("Test Document");

IStorable isDoc = (IStorable) doc;
isDoc.Read();

ICompressible icDoc = (ICompressible) doc;
icDoc.Compress();
```

Если окажется, что данный объект `Document` реализует только интерфейс `IStorable`:

```
public class Document : IStorable
```

то приведение к типу ICompressible все равно будет скомпилировано, поскольку ICompressible является допустимым интерфейсом. Однако из-за недопустимости подобного приведения типа на этапе исполнения будет вызвано исключение:

```
An exception of type System.InvalidCastException was thrown.
```

Исключения подробно обсуждаются в главе 11.

Оператор is

Чтобы вызывать необходимые методы, программист должен иметь возможность выяснить, поддерживает ли объект данный интерфейс. В C# существует два способа узнать это. Первый заключается в применении оператора is. Его синтаксис:

```
выражение is тип
```

Оператор is возвращает значение true, если выражение (которое должно иметь ссылочный тип) может быть безопасно (то есть без вызова исключения) приведено к типу, указанному справа от ключевого слова is, обозначающего операцию. В примере 8.3 показано, как операция is применяется для проверки того, реализует ли класс Document интерфейсы IStorable и ICompressible.

Пример 8.3. Использование операции is

```
using System;

interface IStorable
{
   void Read();
   void Write(object obj);
   int Status { get; set; }

}

// новый интерфейс
interface ICompressible
{
   void Compress();
   void Decompress();
}

// класс Document реализует интерфейс IStorable
public class Document : IStorable
{
   public Document(string s)
   {
      Console.WriteLine(
         "Создание документа: {0}", s);
   }

   // IStorable.Read
```

```
    public void Read()
    {
        Console.WriteLine(
            "Реализация метода Read для IStorable");
    }

    // IStorable.Write
    public void Write(object o)
    {
        Console.WriteLine(
            "Реализация метода Write для IStorable");
    }

    // IStorable.Status
    public int Status
    {
        get
        {
            return status;
        }

        set
        {
            status = value;
        }
    }

    private int status = 0;
}

public class Tester
{
    static void Main()
    {
        Document doc = new Document("Test Document");

        // выполнить только безопасное преобразование типа
        if (doc is IStorable)
        {
            IStorable isDoc = (IStorable) doc;
            isDoc.Read();
        }

        // эта проверка даст отрицательный результат
        if (doc is ICompressible)
        {
            ICompressible icDoc = (ICompressible) doc;
            icDoc.Compress();
        }
    }
}
```

Пример 8.3 отличается от примера 8.2 тем, что в нем класс `Document` не реализует интерфейс `ICompressible`. Метод `Main()` определяет, допусти-

мо ли (то есть безопасно) преобразование типа. С этой целью вычисляется выражение в операторе `if`:

```
if (doc is IStorable)
```

Получился понятный и почти самодокументирующийся код. Оператор `if` прямо говорит о том, что приведение типа будет выполнено, только если объект имеет необходимый тип интерфейса.

К сожалению, такое применение оператора `is` оказывается неэффективным. Чтобы понять почему, необходимо разобраться в MSIL-коде, сгенерированном в этом случае. Вот небольшой отрывок из него (обратите внимание, что номера строк даны в шестнадцатеричной системе счисления):

```
IL_0023:  isinst     ICompressible
IL_0028:  brfalse.s  IL_0039
IL_002a:  ldloc.0
IL_002b:  castclass  ICompressible
IL_0030:  stloc.2
IL_0031:  ldloc.2
IL_0032:  callvirt   instance void ICompressible::Compress()
```

Здесь самой важной является строка 23, где проверяется тип `ICompressible`. Ключевое слово `isinst` является MSIL-кодом для операции `is`. Эта команда проверяет, имеет ли объект (`doc`) указанный тип. Вскоре после этой проверки выполняется команда `castclass` в строке 2b. К сожалению, она тоже проверяет тип объекта. Итак, тип объекта проверяется дважды. Более эффективное решение – применение оператора `as`.

Оператор as

Оператор `as` сочетает в себе оператор `is` и приведение типа. Он сперва проверяет допустимость требуемого преобразования (то есть возвратит ли оператор `is` значение `true`), а затем выполняет приведение типа, если оно безопасно. В противном случае (когда операция `is` возвратила бы `false`) оператор `as` возвращает значение `null`.

Ключевое слово `null` представляет пустую ссылку, то есть ссылку, которая не указывает ни на какой объект.

Применение оператора `as` позволяет обойтись без обработки исключений. В то же время удается избежать двойной проверки допустимости приведения типа. По этим причинам предпочтительнее выполнять приведение с помощью оператора `as`. Его синтаксис:

```
выражение as тип
```

В следующем фрагменте программы метод `Main()` из примера 8.3 изменен так, что в нем используется оператор `as` и проводится проверка на `null`:

```
static void Main()
{
    Document doc = new Document("Test Document");
    IStorable isDoc = doc as IStorable;
    if (isDoc != null)
        isDoc.Read();
    else
        Console.WriteLine("IStorable не поддерживается");

    ICompressible icDoc = doc as ICompressible;
    if (icDoc != null)
        icDoc.Compress();
    else
        Console.WriteLine("Compressible не поддерживается");
}
```

Даже беглый взгляд на MSIL-код позволяет заметить, что теперь он намного эффективнее:

```
IL_0023:  isinst     ICompressible
IL_0028:  stloc.2
IL_0029:  ldloc.2
IL_002a:  brfalse.s  IL_0034
IL_002c:  ldloc.2
IL_002d:  callvirt   instance void ICompressible::Compress()
```

Сравнение операторов is и as

Если в программе проверка типа объекта немедленно сопровождается преобразованием типа (при положительном результате), то оператор `as` более эффективен. Однако в некоторых случаях приведение типа не выполняется сразу после проверки. Возможно, логика программы вообще не требует преобразования типа, например, когда составляется список объектов, поддерживающих некоторый интерфейс. В этом случае лучше применить оператор `is`.

Сравнение интерфейса и абстрактного класса

Интерфейсы во многом схожи с абстрактными классами. Совсем нетрудно изменить объявление интерфейса `IStorable` так, что он превратится в абстрактный класс:

```
abstract class Storable
{
  abstract public void Read();
  abstract public void Write();
}
```

Теперь класс `Document` может быть производным от класса `Storable`, и это наследование будет ненамного отличаться от использования интерфейса.

Но не все так просто. Представим, что некий программист приобрел класс List и желает скомбинировать его возможности с возможностями класса Storable. В C++ можно создать класс StorableList, производный как от List, так и от Storable. Однако в C# это невозможно. Класс не может быть производным одновременно от абстрактного класса Storable и от класса List, поскольку в этом языке запрещено множественное наследование.

Зато C# разрешает реализовать несколько интерфейсов и произвести несколько классов от одного базового. Таким образом, превратив класс Storable в интерфейс, можно создать класс, являющийся производным как от класса List, так и от интерфейса IStorable. Пример – класс StorableList из следующего фрагмента программы:

```
public class StorableList : List, IStorable
{
   // методы класса ...
   public void Read() {...}
   public void Write(object obj) {...}
   // ...
}
```

Переопределение реализации интерфейса

В классе, реализующем интерфейс, любой метод интерфейса (или даже все) может быть объявлен как виртуальный. Производные классы вправе переопределять такие методы (используя ключевое слово override) или предлагать новые реализации (с помощью ключевого слова new). Пусть, например, класс Document реализует интерфейс IStorable и помечает методы Read() и Write() как виртуальные. С помощью этих методов класс Document читает данные из объекта File и, соответственно, пишет в него. Впоследствии программист может наследовать от класса Document другие классы, например Note или EmailMessage, решая при этом, что для чтения и записи информации объект Note будет обращаться к базе данных, а не к файлу.

Пример 8.4, лишенный сложности примера 8.3, иллюстрирует переопределение реализации интерфейса. Класс Document помечает метод Read() ключевым словом virtual и реализует его. Впоследствии метод Read() перегружается в классе Note, произведенном от Document.

Пример 8.4. Переопределение реализации интерфейса

```
using System;

interface IStorable
{
   void Read();
   void Write();
}
```

```
// класс Document упрощен и реализует только интерфейс IStorable
public class Document : IStorable
{
   // конструктор класса Document
   public Document(string s)
   {
      Console.WriteLine(
         "Создание документа: {0}", s);
   }

   // метод Read() объявлен виртуальным
   public virtual void Read()
   {
      Console.WriteLine(
         "Метод Read класса Document для IStorable");
   }

   // Внимание: метод не виртуальный!
   public void Write()
   {
      Console.WriteLine(
         "Метод Write класса Document для IStorable");
   }

}

// произвести класс от класса Document
public class Note : Document
{
   public Note(string s):
      base(s)
   {
      Console.WriteLine(
         "Создание заметки: {0}", s);
   }

   // переопределение метода Read()
   public override void Read()
   {
      Console.WriteLine(
         "Переопределение метода Read для Note!");
   }

   // реализация собственного метода Write()
   public new void Write()
   {
      Console.WriteLine(
         "Реализация метода Write для Note!");
   }
}
public class Tester
{

   static void Main()
```

```
    {
        // создание объекта Document
        Document theNote = new Note("Test Note");
        IStorable isNote = theNote as IStorable;
        if (isNote != null)
        {
            isNote.Read();
            isNote.Write();
        }

        Console.WriteLine("\n");

        // прямой вызов всех методов
        theNote.Read();
        theNote.Write();

        Console.WriteLine("\n");

        // создание объекта Note
        Note note2 = new Note("Second Test");
        IStorable isNote2 = note2 as IStorable;
        if (isNote != null)
        {
            isNote2.Read();
            isNote2.Write();
        }

        Console.WriteLine("\n");

        // прямой вызов всех методов
        note2.Read();
        note2.Write();
    }
}
```

Вывод:

```
Создание документа: Test Note
Создание заметки: Test Note
переопределение метода Read для Note!
Метод Write класса Document для IStorable

Переопределение метода Read для Note!
Метод Write класса Document для IStorable

Создание документа: Second Test
Создание заметки: Second Test
Переопределение метода Read для Note!
Метод Write класса Document для IStorable

Переопределение метода Read для Note!
Реализация метода Write для Note!
```

В этом примере класс `Document` реализует упрощенный интерфейс `IStorable` (упрощение сделано ради ясности примера):

```
interface IStorable
{
```

```
        void Read();
        void Write();
    }
```

В классе Document виртуальным объявлен только метод Read(), но не Write():

```
public virtual void Read()
```

В реальном приложении оба метода были бы виртуальными, но сейчас автор желает продемонстрировать, что разработчик волен выбирать, какие методы будут виртуальными.

Новый класс Note произведен от класса Document:

```
public class Note : Document
```

Совсем необязательно, чтобы класс Note перегружал метод Read(), но он вправе сделать это:

```
public override void Read()
```

В классе Tester методы Read() и Write() вызываются четырьмя разными способами:

1. Через ссылку базового класса на производный объект
2. Через интерфейс, созданный из ссылки базового класса на производный объект
3. Через производный объект
4. Через интерфейс, созданный из производного объекта

Чтобы выполнить два первых вызова, создается ссылка базового класса Document, и этой ссылке присваивается адрес нового (производного) объекта Note, созданного в куче:

```
Document theNote = new Note("Test Note");
```

Затем создается ссылка на интерфейс, и объект Document приводится оператором as к типу ссылки IStorable:

```
IStorable isNote = theNote as IStorable;
```

Методы Read() и Write() вызываются с помощью этого интерфейса. Результат выполнения программы свидетельствует о том, что обращение к Read() полиморфное, а к Write() — нет. Впрочем, именно этого и следовало ожидать:

```
Переопределение метода Read для Note!
Метод Write класса Document для IStorable
```

Затем методы Read() и Write() вызываются непосредственно из объекта:

```
theNote.Read();
theNote.Write();
```

И опять сработала полиморфная реализация метода:

```
Переопределение метода Read для Note!
Метод Write класса Document для IStorable
```

В обоих случаях вызываются метод Read() класса Note и метод Write() класса Document.

Чтобы убедиться, что это результат перегрузки метода, создадим второй объект Note, присвоив его адрес ссылке на тип Note. Это послужит в качестве иллюстрации последних двух случаев (то есть вызова через производный объект и через интерфейс, созданный из производного объекта):

```
Note note2 = new Note("Second Test");
```

Итак, если тип объекта приведен к ссылке, вызывается переопределенный метод Read(). Однако когда методы вызываются непосредственно для объекта Note:

```
note2.Read();
note2.Write();
```

то результат выполнения программы отражает факт обращения к методу объекта Note, а не к переопределенному методу объекта Document.

```
Переопределение метода Read для Note!
Реализация метода Write для Note!
```

Явная реализация интерфейса

В примерах, которые до сих пор были представлены в этой главе, реализующий интерфейс класс (в данном случае Document) создавал метод с той же сигнатурой и возвращаемым значением, что и у метода, названного в интерфейсе. Не было необходимости явно указывать, что это реализация интерфейса, – компилятор разбирался в контексте.

Однако что произойдет, если класс реализует два интерфейса, имеющих методы с совпадающими сигнатурами? В примере 8.5 создаются два интерфейса, IStorable и ITalk. Последний содержит метод Read(), который озвучивает текст. К сожалению, этот метод конфликтует с методом Read() интерфейса IStorable.

Из-за того что оба интерфейса имеют методы Read(), реализующий класс Document должен применить *явную реализацию* (*explicit implementation*), по меньшей мере, для одного из методов. С помощью явной реализации класс (Document) однозначно идентифицирует интерфейс для метода:

```
void ITalk.Read()
```

Это разрешает конфликт, но порождает целый ряд побочных эффектов.

Во-первых, отпадает необходимость в явной реализации второго метода этого интерфейса (то есть метода Talk):

```
public void Talk()
```

Поскольку здесь конфликт отсутствует, метод можно объявлять как обычно.

Более важен тот факт, что явно реализуемый метод не может иметь модификатора доступа:

```
void ITalk.Read()
```

Этот метод открыт по умолчанию.

Вообще, метод, объявленный при явной реализации, не может иметь модификаторов abstract, virtual, override и new.

И самое главное. К явно реализованному методу нельзя обращаться через объект. Если написать:

```
theDoc.Read();
```

то компилятор предположит, что имеется в виду неявно реализованный метод интерфейса IStorable. Единственно возможный способ обратиться к явно реализованному интерфейсу – выполнить приведение типа объекта к типу интерфейса:

```
ITalk itDoc = theDoc as ITalk;
if (itDoc != null)
{
    itDoc.Read();
}
```

Явная реализация продемонстрирована в примере 8.5.

Пример 8.5. Явная реализация интерфейса

```
using System;

interface IStorable
{
    void Read();
    void Write();
}

interface ITalk
{
    void Talk();
    void Read();
}

// класс Document изменен - реализует интерфейсы IStorable и ITalk
public class Document : IStorable, ITalk
{
```

```
     // конструктор класса Document
    public Document(string s)
    {
         Console.WriteLine("Создание документа: {0}", s);
    }

    // метод Read() объявлен виртуальным
    public virtual void Read()
    {
         Console.WriteLine("Реализация IStorable.Read");
    }

    public void Write()
    {
         Console.WriteLine("Реализация IStorable.Write");
    }

    void ITalk.Read()
    {
        Console.WriteLine("Реализация ITalk.Read");
    }
    public void Talk()
    {
        Console.WriteLine("Реализация ITalk.Talk");
    }

}

public class Tester
{

    static void Main()
    {
        // создание объекта Document
        Document theDoc = new Document("Test Document");
        IStorable isDoc = theDoc as IStorable;
        if (isDoc != null)
        {
            isDoc.Read();
        }

        ITalk itDoc = theDoc as ITalk;
        if (itDoc != null)
        {
            itDoc.Read();
        }

        theDoc.Read();
        theDoc.Talk();

    }

}
```

Вывод:

```
Создание документа: Test Document
```

```
Реализация IStorable.Read
Реализация ITalk.Read

Реализация IStorable.Read
Реализация ITalk.Talk
```

Выборочное открытие методов интерфейса

Разработчик класса вправе воспользоваться тем фактом, что интерфейс, реализованный явно, не виден клиентам реализующего класса иначе, чем через приведение типов.

Предположим, что семантика объекта Document требует реализации интерфейса IStorable, но программист не хочет, чтобы методы Read() и Write() были частью открытого интерфейса класса Document. Чтобы гарантировать, что эти методы будут доступны исключительно через приведение типов, программист может применить явную реализацию. Это позволит сохранить семантику класса Document и при этом реализовать интерфейс IStorable. Если в пользовательской программе потребуется объект, реализующий интерфейс IStorable, она может выполнить явное преобразование типа, однако при обращении к документу *как к объекту* Document его семантика не будет включать в себя методы Read() и Write().

В принципе, программист в состоянии выбирать, какие методы сделать видимыми за счет явной реализации. Тогда одни будут видны как часть класса Document, а другие – нет. В примере 8.5 объект Document представляет метод Talk() как метод класса Document, а метод ITalk.Read() можно вызвать только через приведение типа. Даже если бы у интерфейса IStorable не было метода Read(), программист мог бы явно реализовать метод Read() и не предоставлять его как метод класса Document.

Обратите внимание на следующий факт. Поскольку явная реализация интерфейса не позволяет применять ключевое слово virtual, производный класс будет вынужден заново реализовать метод. То есть если класс Note произведен от класса Document, ему придется снова реализовать метод ITalk.Read(), поскольку реализация этого метода классом Document не может быть виртуальной.

Сокрытие членов

Член интерфейса может быть скрыт. Например, пусть имеется интерфейс IBase со свойством P:

```
interface IBase
{
   int P { get; set; }
}
```

Пусть также от этого интерфейса произведен новый, IDerived, который скрывает свойство P своим методом P():

```
interface IDerived : IBase
{
   new int P();
}
```

Не вдаваясь в обсуждение, насколько это удачная идея, отметим, что теперь свойство P скрыто в базовом интерфейсе. Реализация производного интерфейса потребует, чтобы как минимум один элемент был реализован явно. Это может быть либо свойство базового интерфейса, либо метод производного, либо оба. Иными словами, допустимы все три варианта:

```
class myClass : IDerived
{
   // явная реализация свойства базового интерфейса
   int IBase.P { get {...} }

   // неявная реализация метода производного интерфейса
   public int P() {...}
}

 class myClass : IDerived
{
   // неявная реализация свойства базового интерфейса
   public int P { get {...} }

   // явная реализация метода производного интерфейса
   int IDerived.P() {...}
}

class myClass : IDerived
{
   // явная реализация свойства базового интерфейса
   int IBase.P { get {...} }

   // явная реализация метода производного интерфейса
   int IDerived.P() {...}
}
```

Доступ к изолированным классам и размерным типам

Вообще говоря, предпочтительнее обращаться к методам интерфейса через приведение типов. Исключением из этого правила являются размерные типы (например, структуры) и изолированные классы. Для них лучше вызывать методы интерфейса через объект.

Когда программист реализует интерфейс в структуре, реализация имеет размерный тип. При последующем приведении объекта к типу ссылки на интерфейс имеет место неявная упаковка объекта. К сожа-

лению, если попытаться модифицировать объект с помощью этого интерфейса, то модифицирован будет упакованный объект, а не оригинальный. Более того, если изменить размерный тип, то упакованный тип останется неизменным. В примере 8.6 создается структура, которая реализует интерфейс IStorable. После этого демонстрируется действие неявной упаковки, когда структура приводится к типу ссылки на интерфейс.

Пример 8.6. Ссылки на размерные типы

```
using System;

// объявление простого интерфейса
interface IStorable
{
   void Read();
   int Status { get;set;}

}
// реализация в виде структуры
public struct myStruct : IStorable
{

   public void Read()
   {
      Console.WriteLine(
         "Реализация IStorable.Read");
   }

   public int Status
   {
      get
      {
         return status;
      }
      set
      {
         status = value;
      }
   }

   private int status;
}

public class Tester
{

   static void Main()
   {
      // создание объекта myStruct
      myStruct theStruct = new myStruct();
      theStruct.Status = -1;  // инициализация
      Console.WriteLine(
         "theStruct.Status: {0}", theStruct.Status);
```

```
        // изменение значения
        theStruct.Status = 2;
        Console.WriteLine("Изменение через объект.");
        Console.WriteLine(
           "theStruct.Status: {0}", theStruct.Status);

        // приведение к типу IStorable
        // неявная упаковка - преобразование в тип ссылки
        IStorable isTemp = (IStorable) theStruct;

        // установка значения через ссылку на интерфейс
        isTemp.Status = 4;
        Console.WriteLine("Изменение через интерфейс.");
        Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
           theStruct.Status, isTemp.Status);

        // еще одно изменение значения
        theStruct.Status = 6;
        Console.WriteLine("Изменение через объект.");
        Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
           theStruct.Status, isTemp.Status);
    }
}
```

Вывод:

```
theStruct.Status: -1
Изменение через объект.
theStruct.Status: 2
Изменение через интерфейс.
theStruct.Status: 2, isTemp: 4
Изменение через объект.
theStruct.Status: 6, isTemp: 4
```

В примере 8.6 интерфейс `IStorable` содержит метод `Read()` и свойство `Status`.

Интерфейс реализуется в структуре по имени `myStruct`:

```
public struct myStruct : IStorable
```

Класс `Tester` содержит довольно интересный программный код. Он начинается с создания экземпляра структуры и инициализации свойства значением -1. После этого значение свойства `Status` выводится на консоль:

```
myStruct theStruct = new myStruct();
theStruct.status = -1;  // инициализация
Console.WriteLine(
   "theStruct.Status: {0}", theStruct.status);
```

Выводимая на экран строка свидетельствует о том, что инициализация прошла удачно:

```
theStruct.Status: -1
```

Затем программа обращается к свойству, и снова через объект:

```
// изменение значения
theStruct.status = 2;
Console.WriteLine("Изменение через объект.");
Console.WriteLine(
   "theStruct.Status: {0}", theStruct.status);
```

Выводимые строки отражают факт изменения значения:

```
Изменение через объект.
theStruct.Status: 2
```

Пока никаких сюрпризов. Теперь создается ссылка на интерфейс `IStorable`, что приводит к неявной упаковке объекта `theStruct` (имеющего размерный тип). Совершается попытка изменить значение свойства `Status` на 4 через интерфейс:

```
// приведение к типу IStorable
// неявная упаковка - преобразование в тип ссылки
IStorable isTemp = (IStorable) theStruct;

// установка значения через интерфейсную ссылку
isTemp.status = 4;
Console.WriteLine("Изменение через интерфейс.");
Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
    theStruct.status, isTemp.status);
```

Здесь выводимый текст может несколько удивить:

```
Изменение через интерфейс.
theStruct.Status: 2, isTemp: 4
```

Вот так! Объект, на который указывает ссылка на интерфейс, изменил значение на 4, однако объект структуры остался прежним. Что еще интереснее, если вызвать метод непосредственно через объект:

```
// еще одно изменение значения
theStruct.status = 6;
Console.WriteLine("Изменение через объект.");
Console.WriteLine("theStruct.Status: {0}, isTemp: {1}",
    theStruct.status, isTemp.status);
```

то, судя по выводу, изменится объект, но не упакованное значение ссылочного типа:

```
Изменение через объект.
theStruct.Status: 6, isTemp: 4
```

Беглый взгляд на MSIL-код (пример 8.7) позволяет понять, что происходит «за сценой».

Пример 8.7. MSIL-код, сгенерированный для кода из примера 8.6

```
.method private hidebysig static void Main() cil managed
{
```

```
.entrypoint
// Code size       187 (0xbb)
.maxstack  4
.locals init ([0] valuetype myStruct theStruct,
         [1] class ConsoleApplication1.IStorable isTemp)
IL_0000:  ldloca.s   theStruct
IL_0002:  initobj    myStruct
IL_0008:  ldloca.s   theStruct
IL_000a:  ldc.i4.m1
IL_000b:  call       instance void myStruct::set_Status(int32)
IL_0010:  ldstr      "theStruct.Status: {0}"
IL_0015:  ldloca.s   theStruct
IL_0017:  call       instance int32 myStruct::get_Status()
IL_001c:  box        [mscorlib]System.Int32
IL_0021:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object)
IL_0026:  ldloca.s   theStruct
IL_0028:  ldc.i4.2
IL_0029:  call       instance void myStruct::set_Status(int32)
IL_002e:  ldstr      "Изменение через объект."
IL_0033:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0038:  ldstr      "theStruct.Status: {0}"
IL_003d:  ldloca.s   theStruct
IL_003f:  call       instance int32 myStruct::get_Status()
IL_0044:  box        [mscorlib]System.Int32
IL_0049:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object)
IL_004e:  ldloc.0
IL_004f:  box        myStruct
IL_0054:  stloc.1
IL_0055:  ldloc.1
IL_0056:  ldc.i4.4
IL_0057:  callvirt   instance void IStorable::set_Status(int32)
IL_005c:  ldstr      "Изменение через интерфейс."
IL_0061:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0066:  ldstr      "theStruct.Status: {0}, isTemp: {1}"
IL_006b:  ldloca.s   theStruct
IL_006d:  call       instance int32 myStruct::get_Status()
IL_0072:  box        [mscorlib]System.Int32
IL_0077:  ldloc.1
IL_0078:  callvirt   instance int32 IStorable::get_Status()
IL_007d:  box        [mscorlib]System.Int32
IL_0082:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object,
                                                              object)
IL_0087:  ldloca.s   theStruct
IL_0089:  ldc.i4.6
IL_008a:  call       instance void myStruct::set_Status(int32)
IL_008f:  ldstr      "Изменение через объект."
IL_0094:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0099:  ldstr      "theStruct.Status: {0}, isTemp: {1}"
```

```
    IL_009e:  ldloca.s   theStruct
    IL_00a0:  call       instance int32 myStruct::get_Status()
    IL_00a5:  box        [mscorlib]System.Int32
    IL_00aa:  ldloc.1
    IL_00ab:  callvirt   instance int32 IStorable::get_Status()
    IL_00b0:  box        [mscorlib]System.Int32
    IL_00b5:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                  object,
                                                                  object)
    IL_00ba:  ret
  } // end of method Tester::Main
```

В строчке IL_000b вызывается метод set_Status для объекта размерного типа. Еще один вызов можно видеть в строчке IL_0017. Обратите внимание, что вызовы метода WriteLine() влекут за собой упаковку целого значения Status, чтобы можно было вызвать метод GetString().

Очень важной является строчка IL_0056 (выделена полужирным шрифтом), где упаковывается сама структура. Именно процесс упаковки создает ссылочный тип для ссылки на интерфейс. Обратите внимание, что в строчке IL_005e вызывается IStorable::set_status, а не myStruct::setStatus.

Итак, можно сформулировать рекомендацию по программированию: при реализации интерфейса, имеющего размерный тип, следует обращаться к элементам интерфейса через объект, а не через ссылку на интерфейс.

9

Массивы, индексаторы и классы коллекций

Платформа .NET Framework предоставляет программисту богатый набор классов коллекций, в том числе `Array`, `ArrayList`, `NameValueCollection`, `StringCollection`, `Queue`, `Stack` и `BitArray`.

Простейшим классом коллекции является `Array` – единственный, для которого в C# имеется встроенная поддержка. В этой главе показано, как работать с одномерными, многомерными и невыровненными (jagged) массивами. Читатель также узнает, что такое индексаторы, синтаксическая изюминка C#, позволяющая обращаться к свойствам класса так, словно класс индексирован подобно массиву.

Платформа .NET Framework содержит ряд интерфейсов, таких как `IEnumerable` и `ICollection`, реализация которых предоставляет программисту стандартный способ взаимодействия с классами коллекций. В настоящей главе будет продемонстрировано, как работать с самыми важными из этих интерфейсов. Заканчивается глава обзором наиболее популярных классов коллекций платформы .NET, в том числе `ArrayList`, `Dictionary`, `Hashtable`, `Queue` и `Stack`.

Массивы

Массив – это индексированный набор объектов одного типа. В языке C# массивы несколько отличаются от массивов в C++ и других языках, поскольку они являются объектами, что наделяет их полезными методами и свойствами.

В C# определен синтаксис объявления объектов типа `Array`. Однако фактически создается объект типа `System.Array`. Таким образом, C#

предоставляет программисту идеальные условия: за простым синтаксисом в стиле C кроется объявление класса, дающее экземплярам массива доступ к методам и свойствам класса `System.Array`. Все они перечислены в табл. 9.1.

Таблица 9.1. Методы и свойства класса System.Array

Метод или свойство	Описание
`BinarySearch()`	Перегруженный открытый статический метод, выполняющий поиск в одномерном упорядоченном массиве
`Clear()`	Открытый статический метод, который приравнивает диапазон элементов массива к нулю или к нулевой ссылке
`Copy()`	Перегруженный открытый статический метод, копирующий фрагмент одного массива в другой
`CreateInstance()`	Перегруженный открытый статический метод, создающий новый экземпляр массива
`IndexOf()`	Перегруженный открытый статический метод, возвращающий индекс (смещение от начала) первого значения в одномерном массиве
`LastIndexOf()`	Перегруженный открытый статический метод, возвращающий индекс последнего значения в одномерном массиве
`Reverse()`	Перегруженный открытый статический метод, меняющий порядок следования элементов одномерного массива на обратный
`Sort()`	Перегруженный открытый статический метод, сортирующий элементы одномерного массива
`IsFixedSize`	Открытое свойство, возвращающее логическое значение, которое сообщает, фиксирован ли размер массива
`IsReadOnly`	Открытое свойство, возвращающее логическое значение, которое сообщает, доступен ли массив только для чтения
`IsSynchronized`	Открытое свойство, возвращающее логическое значение, которое сообщает, обеспечивает ли массив безопасную работу с несколькими потоками
`Length`	Открытое свойство, возвращающее длину массива
`Rank`	Открытое свойство, возвращающее число измерений массива
`SyncRoot`	Открытое свойство, возвращающее объект, с помощью которого можно синхронизировать доступ к массиву
`GetEnumerator()`	Открытый метод, возвращающий объект типа `IEnumerator`
`GetLength()`	Открытый метод, возвращающий длину указанного измерения массива
`GetLowerBound()`	Открытый метод, возвращающий нижнюю границу указанного измерения массива

Метод или свойство	Описание
GetUpperBound()	Открытый метод, возвращающий верхнюю границу указанного измерения массива
Initialize()	Инициализирует все элементы массива (имеющие размерный тип), вызывая для каждого конструктор по умолчанию
SetValue()	Перегруженный открытый метод, присваивающий значение указанному элементу массива

Объявление массивов

Объявление массивов имеет следующий синтаксис:

```
тип[] имя-массива;
```

Например:

```
int[] myIntArray;
```

Квадратные скобки ([ ]) сообщают компилятору C#, что определяется массив, а тип указывает тип элементов этого массива. В примере, приведенном выше, myIntArray является целочисленным массивом.

Экземпляр массива создается ключевым словом new. Например:

```
myIntArray = new int[5];
```

Это объявление отводит память для массива, состоящего из пяти целых чисел.

Вниманию программистов, пишущих на языке Visual Basic! Первый элемент массива всегда имеет индекс 0; нельзя установить верхнюю и нижнюю границы массива, а также изменить размер массива (оператор redim отсутствует).

Важно проводить различие между собственно массивом (который представляет собой набор объектов) и его элементами. В рассматриваемом примере myIntArray является массивом, а его элементы – пять целых чисел. В языке C# массивы относятся к ссылочным типам, объекты которых создаются в куче. Соответственно, массив myIntArray находится в куче. Элементы массива размещаются в соответствии с их типами. Целочисленный тип относится к размерным типам, поэтому элементы массива myIntArray будут иметь размерный тип, а *не* упакованный целочисленный тип. Массив элементов, имеющих ссылочный тип, будет содержать только ссылки на элементы, размещенные в куче.

Значения по умолчанию

Когда создается массив элементов, имеющих тип значения, каждый элемент первоначально содержит значение, устанавливаемое по умол-

чанию для данного типа (см. табл. 4.2 «Базовые типы и их значения по умолчанию»). Объявление:

```
myIntArray = new int[5];
```

создает массив из пяти целых, причем каждому элементу присваивается нулевое значение, устанавливаемое по умолчанию для переменных типа int.

В отличие от массивов с элементами размерных типов, элементы массива, имеющие ссылочный тип, не инициализируются значениями по умолчанию. Вместо этого им присваивается значение null. Если попытаться обратиться к элементу массива, элементы которого имеют ссылочный тип до его явной инициализации, будет вызвано исключение.

Предположим, имеется класс Button. Массив объектов типа Button объявляется следующим образом:

```
Button[] myButtonArray;
```

а экземпляр массива создается так:

```
myButtonArray = new Button[3];
```

Эти два оператора можно объединить:

```
Button myButtonArray = new Button[3];
```

В отличие от предыдущего примера с массивом целых, этот оператор не создает массив ссылок на три объекта Button. Вместо этого создается массив myButtonArray из трех нулевых (null) ссылок. Чтобы им можно было пользоваться, вначале нужно создать и инициализировать объекты Button для каждой ссылки из массива. Создавать объекты можно в цикле, добавляя их в массив по одному.

Обращение к элементам массива

Программист обращается к элементам любого массива с помощью операции индексирования ([]). Нумерация элементов массива начинается с нуля, иными словами, индекс первого элемента массива всегда имеет нулевое значение. В рассматриваемом случае первый элемент – myArray[0].

Как уже говорилось, массивы являются объектами, и поэтому у них есть свойства. Одно из наиболее полезных свойств – Length, которое сообщает, сколько объектов содержится в массиве. Объекты массива индексируются от 0 до Length-1. То есть если в массиве пять элементов, то их индексы будут: 0, 1, 2, 3, 4.

В примере 9.1 иллюстрируется все, сказанное выше. В этой программе класс Tester создает массив элементов типа Employee и массив целых чисел. Массив Employee заполняется значениями, после чего выводятся элементы обоих массивов.

Пример 9.1. Работа с массивом

```
namespace Programming_CSharp
{
   using System;

   // простой класс для хранения в массиве
   public class Employee
   {
      // простой класс для хранения в массиве
      public Employee(int empID)
      {
         this.empID = empID;
      }
      public override  string ToString()
      {
         return empID.ToString();
      }
      private int empID;

      private int size;
   }
   public class Tester
   {
      static void Main()
      {
         int[] intArray;
         Employee[] empArray;
         intArray = new int[5];
         empArray = new Employee[3];

         // заполнение массива
         for (int i = 0;i<empArray.Length;i++)
         {
            empArray[i] = new Employee(i+5);
         }

         for (int i = 0;i<intArray.Length;i++)
         {
            Console.WriteLine(intArray[i].ToString());
         }

         for (int i = 0;i<empArray.Length;i++)
         {
            Console.WriteLine(empArray[i].ToString());
         }
      }
   }
}
```

Вывод:

```
0
0
0
```

```
0
0
5
6
7
```

Пример начинается с определения класса Employee, конструктор которого принимает один параметр – целое число. Метод ToString(), унаследованный от класса Object, переопределяется так, что выводит значение свойства empID объекта Employee.

Тестирующий метод объявляет пару массивов и создает их экземпляры. Массив целых автоматически заполняется целыми элементами, имеющими значение 0. Массив Employee должен быть заполнен вручную.

В конце программы содержимое массивов выводится на экран для проверки того, что они были заполнены так, как и было задумано. Вначале выводятся пять целых значений, а за ними – три объекта типа Employee.

Оператор foreach

Оператор цикла foreach является новым для семейства языков C, однако он хорошо знаком программистам, пишущим на VB. Этот оператор позволяет перебирать все элементы массива или иного класса коллекций, обращаясь к каждому из них по очереди. Синтаксис оператора foreach такой:

```
foreach (тип идентификатор in выражение) оператор
```

Таким образом, пример 9.1 можно модифицировать, заменив операторы for, перебирающие элементы массива, на foreach. Новая версия представлена в примере 9.2.

Пример 9.2. Применение оператора foreach

```
namespace Programming_CSharp
{
   using System;

   // простой класс для хранения в массиве
   public class Employee
   {
      // простой класс для хранения в массиве
      public Employee(int empID)
      {
         this.empID = empID;
      }
      public override  string ToString()
      {
         return empID.ToString();
      }
      private int empID;
```

```
        private int size;
    }
    public class Tester
    {
        static void Main()
        {
            int[] intArray;
            Employee[] empArray;
            intArray = new int[5];
            empArray = new Employee[3];

            // заполнение массива
            for (int i = 0;i<empArray.Length;i++)
            {
                empArray[i] = new Employee(i+10);
            }

            foreach (int i in intArray)
            {
                Console.WriteLine(i.ToString());
            }

            foreach (Employee e in empArray)
            {
                Console.WriteLine(e.ToString());
            }
        }
    }
}
```

Результат работы программы, приведенной в примере 9.2, идентичен результату работы программы, приведенной в примере 9.1. Зато теперь нет оператора `for`, измеряющего длину массива и использующего временную переменную в качестве индекса:

```
for (int i = 0; i < empArray.Length; i++)
{
    Console.WriteLine(empArray[i].ToString());
}
```

Вместо него выполняется перебор элементов массива с помощью оператора `foreach`, который автоматически извлекает очередной элемент массива и присваивает его временному объекту, созданному в заголовке оператора.

```
foreach (Employee e in empArray)
{
   Console.WriteLine(e.ToString());
}
```

Объект, извлекаемый из массива, имеет корректный тип, то есть для него можно вызвать любой открытый метод.

Инициализация элементов массива

Содержимое массива можно инициализировать в момент создания экземпляра массива, указав в фигурных скобках список начальных значений. В языке C# существует два синтаксических варианта этого действия, полный и сокращенный:

```
int[] myIntArray = new int[5] { 2, 4, 6, 8, 10 }
int[] myIntArray = { 2, 4, 6, 8, 10 }
```

Между ними нет никакой разницы, и большинство программистов предпочитают короткий вариант, потому что человек по природе ленив. Достаточно ленив, чтобы работать целый день ради экономии нескольких минут при выполнении какого-нибудь действия (что, впрочем, не так уж глупо, если приходится повторять это действие сотни раз).

Ключевое слово params

Нетрудно создать метод, который выводит на экран произвольное количество целых чисел, если передать ему целочисленный массив и затем перебирать их в цикле `foreach`. Однако ключевое слово `params` позволяет передавать методу переменное количество параметров без обязательного явного создания массива.

В следующем примере создается метод `DisplayVals()`, который принимает переменное количество целых аргументов:

```
public void DisplayVals(params int[] intVals)
```

Метод обрабатывает эту конструкцию так, словно массив целых был явно создан и передан в качестве аргумента. Элементы такого массива можно перебрать в цикле, как и элементы любого другого целочисленного массива:

```
foreach (int i in intVals)
{
   Console.WriteLine("DisplayVals {0}",i);
}
```

При этом вызывающий метод вовсе не обязан создавать массив явно. Будет достаточно, если он передаст целые числа, а компилятор соберет аргументы в массив для метода `DisplayVals()`:

```
t.DisplayVals(5,6,7,8);
```

Впрочем, если программист предпочтет передать массив, ничто не помешает сделать это:

```
int [] explicitArray = new int[5] {1,2,3,4,5};
t.DisplayVals(explicitArray);
```

В примере 9.3 приведена программа, иллюстрирующая применение ключевого слова `params`.

Пример 9.3. Иллюстрация применения ключевого слова params

```
namespace Programming_CSharp
{
   using System;

   public class Tester
   {
      static void Main()
      {
         Tester t = new Tester();
         t.DisplayVals(5,6,7,8);
         int [] explicitArray = new int[5] {1,2,3,4,5};
         t.DisplayVals(explicitArray);
      }

      public void DisplayVals(params int[] intVals)
      {
         foreach (int i in intVals)
         {
            Console.WriteLine("DisplayVals {0}",i);
         }
      }
   }
}
```

Вывод:

```
DisplayVals 5
DisplayVals 6
DisplayVals 7
DisplayVals 8
DisplayVals 1
DisplayVals 2
DisplayVals 3
DisplayVals 4
DisplayVals 5
```

Многомерные массивы

Массив можно представлять себе в виде ряда почтовых ящиков, в которые «раскладываются» значения элементов. Вообразите несколько рядов почтовых ящиков, один под другим, как в подъезде многоквартирного дома. Получится классический двухмерный массив со строками и столбцами. Строки идут по горизонтали, столбцы – по вертикали.

Возможно и третье измерение, но для него потребуется чуть больше фантазии. Прикрепим второй двухмерный массив почтовых ящиков поверх того, что висит на стене подъезда, а третий – ко второму и т. д. Хорошо, теперь представьте четыре измерения. Теперь 10.

Если читатель, как и автор этих строк, не силен в теории многомерных пространств, он, должно быть, бросил эти упражнения. Впрочем, мно-

гомерные массивы весьма полезны, независимо от того, может ли программист вообразить, как они выглядят.

C# поддерживает два вида многомерных массивов – прямоугольные и невыровненные. У прямоугольного массива все измерения имеют фиксированный размер. Невыровненный массив по сути своей является массивом массивов, которые могут иметь различную размерность.

Прямоугольные массивы

Прямоугольный массив – это массив, имеющий два (или более) измерения. В классическом двухмерном массиве первое измерение представляет собой число строк, а второе – число столбцов массива.

При объявлении двухмерного массива используется следующий синтаксис:

```
тип [,] имя-массива
```

Например, чтобы объявить и создать экземпляр двухмерного прямоугольного массива по имени `myRectangularArray`, содержащий две строки и три столбца целых чисел, нужно написать:

```
int [,] myRectangularArray = new int[2,3];
```

В примере 9.4 объявляется, инициализируется и выводится на экран двухмерный массив. Инициализация элементов производится в цикле `for`.

Пример 9.4. Прямоугольные массивы

```
namespace Programming_CSharp
{
   using System;

   public class Tester
   {
      static void Main()
      {
         const int rows = 4;
         const int columns = 3;

         // объявление массива целых 4x3
         int[,] rectangularArray = new int[rows, columns];

         // заполнение массива
         for (int i = 0;i < rows;i++)
         {
            for (int j = 0;j<columns;j++)
            {
               rectangularArray[i,j] = i+j;
            }
         }

         // вывести содержимое массива
```

```
            for (int i = 0;i < rows;i++)
            {
               for (int j = 0;j<columns;j++)
               {
                  Console.WriteLine("rectangularArray[{0},{1}] = {2}",
                     i,j,rectangularArray[i,j]);
               }
            }
         }
      }
   }
```

Вывод:

```
rectangularArray[0,0] = 0
rectangularArray[0,1] = 1
rectangularArray[0,2] = 2
rectangularArray[1,0] = 1
rectangularArray[1,1] = 2
rectangularArray[1,2] = 3
rectangularArray[2,0] = 2
rectangularArray[2,1] = 3
rectangularArray[2,2] = 4
rectangularArray[3,0] = 3
rectangularArray[3,1] = 4
rectangularArray[3,2] = 5
```

В этом примере объявляется пара констант:

```
const int rows = 4;
const int columns = 3;
```

Впоследствии с их помощью задаются размерности массива:

```
int[,] rectangularArray = new int[rows, columns];
```

Обратим внимание на синтаксис. Квадратные скобки в конструкции `int[,]` указывают, что объявляется целочисленный массив, а запятая говорит о том, что он двухмерный (две запятых свидетельствуют о трехмерном массиве и т. д.). Создание объекта `rectangularArray` с помощью конструкции `int[rows, columns]` устанавливает размер каждого измерения. Здесь объявление и создание объекта собраны воедино в одном операторе.

Программа заполняет массив в двух циклах `for`, перебирая каждый столбец в каждой строке. Так, в первую очередь, заполняется элемент `rectangularArray[0,0]`, затем `rectangularArray[0,1]`, `rectangularArray[0,2]`. Закончив с первой строкой, программа переходит ко второй: `rectangularArray[1,0]`, `rectangularArray[1,1]`, `rectangularArray[1,2]`. И так далее, пока не будут заполнены все строки.

Вспомним, что одномерный массив инициализируется списком значений в фигурных скобках. Аналогичный синтаксис допускается для

двухмерного массива. В примере 9.5 объявляется двухмерный массив `rectangularArray`. Его элементы инициализируются с помощью списков значений, а затем выводятся на экран.

Пример 9.5. Инициализация многомерного массива

```
namespace Programming_CSharp
{
   using System;

   public class Tester
   {
      static void Main()
      {
         const int rows = 4;
         const int columns = 3;

         // неявно подразумевается массив 4x3
         int[,] rectangularArray =
            {
                {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11}
            };

         for (int i = 0;i < rows;i++)
         {
            for (int j = 0;j<columns;j++)
            {
               Console.WriteLine("rectangularArray[{0},{1}] = {2}",
                  i,j,rectangularArray[i,j]);
            }
         }
      }
   }
}
```

Вывод:

```
rectangularArray[0,0] = 0
rectangularArray[0,1] = 1
rectangularArray[0,2] = 2
rectangularArray[1,0] = 3
rectangularArray[1,1] = 4
rectangularArray[1,2] = 5
rectangularArray[2,0] = 6
rectangularArray[2,1] = 7
rectangularArray[2,2] = 8
rectangularArray[3,0] = 9
rectangularArray[3,1] = 10
rectangularArray[3,2] = 11
```

Этот пример очень похож на пример 9.4, но размерности массива *подразумеваются* способом его заполнения:

```
int[,] rectangularArrayrectangularArray =
{
```

```
    {0,1,2}, {3,4,5}, {6,7,8}, {9,10,11}
};
```

Присваивание значений с помощью четырех списков по три элемента подразумевает массив 4×3.

Если бы было написано:

```
int[,] rectangularArrayrectangularArray =
{
   {0,1,2,3}, {4,5,6,7}, {8,9,10,11}
};
```

то подразумевался бы массив 3×4.

Из выведенного программой текста ясно, что компилятор C# понимает неявное определение размерности, поскольку возможно получить доступ к элементам, указав соответствующее смещение.

Если читатель подумает, что перед ним всего лишь массив из 12 элементов и что обратиться к элементу `rectangularArray[0,3]` так же просто, как к элементу `rectangularArray[1,0]`, это заблуждение будет моментально развеяно вызовом исключения:

```
Exception occurred: System.IndexOutOfRangeException:
Index was outside the bounds of the array.
at Programming_CSharp.Tester.Main() in
csharp\programming csharp\listing0703.cs:line 23
```

Массивы C# достаточно «интеллектуальны», чтобы следить за своими границами. Если подразумевался массив 4×3, то и обращаться с ним следует соответствующим образом.

Невыровненные массивы

Невыровненный массив (*jagged array*) – это массив массивов. Его называют невыровненным, поскольку составляющие его массивы могут иметь разную длину, и тогда графическое представление массива не будет прямоугольным.

Создавая невыровненный массив, программист объявляет в нем число строк. Каждая строка будет содержать массив произвольной длины. Каждый из «внутренних» массивов должен быть объявлен, после чего можно заполнять его элементы.

В невыровненных массивах каждое из измерений представляет собою одномерный массив. Для объявления невыровненного массива используйте следующий синтаксис, в котором количество скобок определяет количество измерений массива:

```
тип [] []...
```

Таким образом, двухмерный невыровненый массив целых чисел `myJaggedArray` определяется так:

```
int [] [] myJaggedArray;
```

Чтобы обратиться к пятому элементу третьего массива, следует написать: `myJaggedArray[2][4]`.

В примере 9.6 создается иерархический массив `myJaggedArray`. Его элементы инициализируются и выводятся на экран. В целях экономии места в программе учитывается тот факт, что элементы массива целых чисел автоматически инициализируются нулями. Поэтому начальные значения присваиваются лишь некоторым элементам.

Пример 9.6. Работа с иерархическим массивом

```
namespace Programming_CSharp
{
   using System;

   public class Tester
   {
      static void Main()
      {
         const int rows = 4;

         // объявление невыровненного массива с 4 строками
         int[][] jaggedArray = new int[rows][];

         // первая строка состоит из 5 элементов
         jaggedArray[0] = new int[5];

         // строка из 2 элементов
         jaggedArray[1] = new int[2];

         // строка из 3 элементов
         jaggedArray[2] = new int[3];

         // последняя строка состоит из 5 элементов
         jaggedArray[3] = new int[5];

         // заполнить некоторые (не все) элементы иерархического массива
         jaggedArray[0][3] = 15;
         jaggedArray[1][1] = 12;
         jaggedArray[2][1] = 9;
         jaggedArray[2][2] = 99;
         jaggedArray[3][0] = 10;
         jaggedArray[3][1] = 11;
         jaggedArray[3][2] = 12;
         jaggedArray[3][3] = 13;
         jaggedArray[3][4] = 14;

         for (int i = 0;i < 5; i++)
         {
            Console.WriteLine("jaggedArray[0][{0}] = {1}",
               i,jaggedArray[0][i]);
         }

         for (int i = 0;i < 2; i++)
         {
```

```
            Console.WriteLine("jaggedArray[1][{0}] = {1}",
               i,jaggedArray[1][i]);
         }

         for (int i = 0;i < 3; i++)
         {
            Console.WriteLine("jaggedArray[2][{0}] = {1}",
               i,jaggedArray[2][i]);
         }
         for (int i = 0;i < 5; i++)
         {
            Console.WriteLine("jaggedArray[3][{0}] = {1}",
               i,jaggedArray[3][i]);
         }
      }
   }
}
```

Вывод:

```
jaggedArray[0][0] = 0
jaggedArray[0][1] = 0
jaggedArray[0][2] = 0
jaggedArray[0][3] = 15
jaggedArray[0][4] = 0
jaggedArray[1][0] = 0
jaggedArray[1][1] = 12
jaggedArray[2][0] = 0
jaggedArray[2][1] = 9
jaggedArray[2][2] = 99
jaggedArray[3][0] = 10
jaggedArray[3][1] = 11
jaggedArray[3][2] = 12
jaggedArray[3][3] = 13
jaggedArray[3][4] = 14
```

В этом примере создается невыровненный массив из 4 строк:

```
int[][] jaggedArray = new int[rows][];
```

Обратите внимание, что второе измерение не указано. Оно устанавливается в каждой строке отдельно при создании нового массива для этой строки. Каждый из таких массивов может иметь свою длину:

```
// первая строка состоит из 5 элементов
jaggedArray[0] = new int[5];

// строка из 2 элементов
jaggedArray[1] = new int[2];

// строка из 3 элементов
jaggedArray[2] = new int[3];

// последняя строка состоит из 5 элементов
jaggedArray[3] = new int[5];
```

Когда массив для каждой строки указан, необходимо заполнить различные элементы этих массивов и вывести их содержимое, чтобы убедиться, что все произошло, как и задумывалось.

Обратите внимание, что при обращении к элементам прямоугольного массива индексы помещаются в одну пару квадратных скобок:

```
rectangularArrayrectangularArray[i,j]
```

в то время как для невыровненного массива требуется несколько пар квадратных скобок:

```
jaggedArray[3][i]
```

Это легко понять, если считать прямоугольный массив единым массивом с несколькими измерениями, а иерархический – массивом массивов.

Преобразования массивов

Преобразование одного массива в другой возможно, если их размерности одинаковы, а также допустимо преобразование типа элементов одного массива в тип элементов другого. Если элементы могут быть преобразованы неявно, выполняется неявное преобразование, в противном случае следует сделать это явно.

Когда массив содержит ссылки на объекты ссылочного типа, возможно преобразование в массив элементов базового типа. Пример 9.7 иллюстрирует преобразование типа `Employee`, определенного пользователем, в массив объектов.

Пример 9.7. Преобразования массивов

```
namespace Programming_CSharp
{
   using System;

   // создать объект, который можно хранить в массиве
   public class Employee
   {
      // простой класс для записи в массив
      public Employee(int empID)
      {
         this.empID = empID;
      }
      public override  string ToString()
      {
         return empID.ToString();
      }
      private int empID;

      private int size;
   }
```

```
   public class Tester
   {
      // этот метод принимает массив элементов типа Object,
      // ему будет передан массив элементов типа Employee,
      // а затем - массив строк
      // преобразование является неявным, поскольку как тип Employee,
      // так и строки произведены (в конечном счете) от класса Object
      public static void PrintArray(object[] theArray)
      {
         Console.WriteLine("Содержимое  Array {0}",
            theArray.ToString());

         // перебрать элементы массива и вывести
         // их значения
         foreach (object obj in theArray)
         {
            Console.WriteLine("Значение: {0}", obj);
         }
      }

      static void Main()
      {
         // создать массив объектов Employee
         Employee[] myEmployeeArray = new Employee[3];
         // инициализировать значение каждого объекта типа Employee
         for (int i = 0;i < 3;i++)
         {
            myEmployeeArray[i] = new Employee(i+5);
         }

         // вывести значения
         PrintArray(myEmployeeArray);

         // создать массив из двух строк
         string[] array =
            {
               "hello", "world"
            };

         // распечатать строки
         PrintArray(array);
      }
   }
}
```

Вывод:

```
Содержимое Array Programming_CSharp.Employee[]
Значение: 5
Значение: 6
Значение: 7
Содержимое Array System.String[]
Значение: hello
Значение: world
```

Пример 9.7 начинается с создания простого класса Employee, как уже делалось ранее в этой главе. Класс Tester содержит теперь новый статический метод PrintArray(), который принимает в качестве параметра одномерный массив элементов типа Object:

```
public static void PrintArray(object[] theArray)
```

Object — это неявный базовый класс для любого объекта платформы .NET Framework, а значит, и для String и для Employee.

Метод PrintArray() предпринимает два действия. Во-первых, он вызывает метод ToString() собственно для массива:

```
Console.WriteLine("Содержимое Array {0}",
     theArray.ToString());
```

Класс System.Array переопределяет метод ToString() так, что тот выводит идентификационное имя массива:

```
Содержимое Array Programming_CSharp. Employee []
Содержимое Array System.String[]
```

Во-вторых, метод PrintArray() вызывает метод ToString() для каждого элемента массива, переданного ему в качестве формального параметра. Поскольку ToString() является виртуальным методом базового класса Object, он гарантированно доступен в каждом производном классе. В рассматриваемом примере этот метод переопределяется в классе Employee, и код работает как надо. Вызов ToString() для объекта String, может, и необязателен, но абсолютно безвреден и позволяет относиться к объектам полиморфно.

Класс System.Array

Класс Array обладает рядом полезных методов, расширяющих возможности массивов и делающих их «интеллектуальнее», чем массивы в других языках (см. табл. 9.1 ранее в этой главе). Двумя наиболее важными методами класса Array являются Sort() и Reverse(). Они полностью поддерживаются для любого базового типа C#, например string. Заставить эти методы работать с пользовательским классом несколько сложнее, поскольку потребуется реализация интерфейса IComparable (см. раздел «Реализация интерфейса IComparable» далее в этой главе). В примере 9.8 демонстрируется применение этих двух методов для обработки строковых объектов.

Пример 9.8. Использование методов Array.Sort и Array.Reverse

```
namespace Programming_CSharp
{
    using System;

    public class Tester
    {
```

```
        public static void PrintMyArray(object[] theArray)
        {

            foreach (object obj in theArray)
            {
                Console.WriteLine("Значение: {0}", obj);
            }
            Console.WriteLine("\n");
        }

        static void Main()
       {
            String[] myArray =
            {
                "Who", "is", "John", "Galt"
            };

            PrintMyArray(myArray);
            Array.Reverse(myArray);
            PrintMyArray(myArray);

            String[] myOtherArray =
            {
                "We", "Hold", "These", "Truths",
                "To", "Be", "Self", "Evident",
            };

            PrintMyArray(myOtherArray);
            Array.Sort(myOtherArray);
            PrintMyArray(myOtherArray);

       }
    }
}
```

Вывод:

```
Значение: Who
Значение: is
Значение: John
Значение: Galt

Значение: Galt
Значение: John
Значение: is
Значение: Who

Значение: We
Значение: Hold
Значение: These
Значение: Truths
Значение: To
Значение: Be
Значение: Self
Значение: Evident
```

```
Значение: Be
Значение: Evident
Значение: Hold
Значение: Self
Значение: These
Значение: To
Значение: Truths
Значение: We
```

Пример начинается с создания массива myArray, массива строк со словами:

```
"Who", "is", "John", "Galt"
```

Массив выводится на экран, а затем передается методу Array.Reverse(), после чего снова выводится для проверки того, изменился ли порядок следования элементов:

```
Значение: Galt
Значение: John
Значение: is
Значение: Who
```

Аналогичным образом создается второй массив, myOtherArray, который содержит слова:

```
"We", "Hold", "These", "Truths",
"To", "Be", "Self", "Evident",
```

Этот массив передается методу Array.Sort(), который благополучно сортирует его в алфавитном порядке:

```
Значение: Be
Значение: Evident
Значение: Hold
Значение: Self
Значение: These
Значение: To
Значение: Truths
Значение: We
```

Индексаторы

В некоторых ситуациях желательно обращаться к объектам классов коллекций, являющихся членами класса, так, словно сам класс является массивом. Предположим, требуется создать элемент управления «окно списка» с именем myListBox. Пусть он содержит список строк, хранящийся в одномерном массиве, а точнее – в закрытой переменной с именем strings. Помимо массива строк окно списка содержит целый ряд свойств и методов. Было бы удобно обращаться к этому элементу

управления, используя индексы, словно окно списка является массивом. Например, появилась бы возможность писать такие операторы:

```
string theFirstString = myListBox[0];
string theLastString = myListBox[Length-1];
```

Индексатор (*indexer*) в C# как раз является конструкцией, позволяющей программисту пользоваться привычным синтаксисом с квадратными скобками для обращения к объектам классов коллекций, являющихся членами класса. Индексатор – это специальный вид свойства, поведение которого определяется методами `get()` и `set()`.

Индексатор объявляется следующим образом:

```
тип this [тип аргумент]{get; set;}
```

Возвращаемый тип определяет тип объекта, возвращаемого индексатором, а тип аргумента определяет, какие аргументы будут задействованы для индексации объекта класса коллекции, содержащего требуемые объекты. Хотя общепринято использовать в качестве индексов целые числа, допустимо индексировать объекты классов коллекций аргументами другого типа, например строками. Более того, можно указать несколько аргументов, создав тем самым многомерный массив!

Ключевое слово `this` – это ссылка на объект, в котором появляется индексатор. Как и в случае обычного свойства, необходимо определить методы `get()` и `set()`, описывающие, как запрошенный объект будет извлекаться из объекта класса коллекции или записываться в него.

В примере 9.9 объявляется элемент управления «список» (`ListBoxTest`). Он содержит простой массив `strings` и индексатор, осуществляющий доступ к содержимому списка.

Вниманию программистов, пишущих на языке C++! Индексатор служит тем же целям, что и перегрузка операции индексирования ([ ]) в языке C++. В C# эта операция не может быть перегружена, что и привело к появлению индексатора.

Пример 9.9. Применение простого индексатора

```
namespace Programming_CSharp
{
   using System;

   // упрощенный элемент управления ListBox
   public class ListBoxTest
   {
      // инициализация списка строками
      public ListBoxTest(params string[] initialStrings)
      {
         // выделение памяти под строки
         strings = new String[256];
```

```
        // копирование строк, переданных конструктору
        foreach (string s in initialStrings)
        {
            strings[ctr++] = s;
        }
    }

    // добавление одиночной строки в конец списка
    public void Add(string theString)
    {
        if (ctr >= strings.Length)
        {
            // обработка недопустимого индекса
        }
        else
            strings[ctr++] = theString;
    }

    // реализация доступа "в стиле массива"
    public string this[int index]
    {
        get
        {
            if (index < 0 || index >= strings.Length)
            {
                // обработка недопустимого индекса
            }
            return strings[index];
        }
        set
        {
            // добавление только с помощью метода Add()
            if (index >= ctr )
            {
                // обработка ошибки
            }
            else
                strings[index] = value;
        }
    }

    // возвращение количества строк в списке
    public int GetNumEntries()
    {
        return ctr;
    }

    private string[] strings;
    private int ctr = 0;
}

public class Tester
{
```

```
        static void Main()
        {
           // создание и инициализация нового списка
           ListBoxTest lbt =
              new ListBoxTest("Hello", "World");

           // добавление в список нескольких строк
           lbt.Add("Who");
           lbt.Add("Is");
           lbt.Add("John");
           lbt.Add("Galt");

           // тестирование доступа
           string subst = "Universe";
           lbt[1] = subst;

           // обращение ко всем строкам
           for (int i = 0;i<lbt.GetNumEntries();i++)
           {
              Console.WriteLine("lbt[{0}]: {1}",i,lbt[i]);
           }
        }
     }
  }
```

Вывод:

```
lbt[0]: Hello
lbt[1]: Universe
lbt[2]: Who
lbt[3]: Is
lbt[4]: John
lbt[5]: Galt
```

Для упрощения примера были отброшены некоторые характеристики реального окна списка, не имеющие отношения к рассматриваемой теме. В примере 9.9 игнорируется все, что связано с поведением списка как элемента пользовательского интерфейса, а внимание сосредоточено на строках и методах, манипулирующих ими. Естественно, в реальном приложении они составили бы лишь малую часть методов окна списка, чье главное предназначение – выводить строки и позволять пользователю выбирать из них.

Первое, на что следует обратить внимание, – два закрытых члена класса:

```
private string[] strings;
private int ctr = 0;
```

В этой программе окно списка содержит одномерный массив строк `strings`. В реальном приложении контейнер для строк был бы сложнее и динамичнее (например, им может быть хеш-таблица, описываемая далее в этой главе). Переменная `ctr` отслеживает количество строк, добавленных в массив.

Инициализация массива выполняется в конструкторе оператором:

```
strings = new String[256];
```

Во второй части реализации конструктора происходит запись формальных параметров в массив. Опять-таки для простоты новые строки добавляются в массив в порядке поступления.

Поскольку количество добавляемых строк заранее неизвестно, аргумент снабжен ключевым словом `params`, рассмотренным ранее в этой главе.

Метод `Add()` класса `ListBoxTest` ограничивается тем, что добавляет новую строку во внутренний массив.

Самым важным методом класса `ListBoxTest` является индексатор. У него нет имени, поэтому приходится пользоваться ключевым словом `this`:

```
public string this[int index]
```

Синтаксис индексатора аналогичен синтаксису свойств. В нем присутствует процедура доступа `get()` или `set()` (или обе). В рассматриваемом примере процедура доступа `get()` выполняет элементарную проверку границ и, если указанный индекс допустим, возвращает запрошенное значение:

```
get
{
    if (index < 0 || index >= strings.Length)
    {
       // обработка недопустимого индекса
    }
    return strings[index];
}
```

Процедура доступа `set()` убеждается, что элемент с указанным индексом уже имеет какое-то значение. В противном случае попытка установить значение считается ошибочной (при таком подходе новые элементы добавляются только методом `Add()`). В процедуре доступа `set()` используется неявный параметр `value`, значение которого присваивается элементу массива с помощью операции индексирования.

```
set
{
if (index >= ctr )
 {
    // обработка ошибки
 }
 else
    strings[index] = value;
}
```

Если теперь написать:

```
lbt[5] = "Hello World"
```

то компилятор вызовет процедуру доступа set() (процедуру доступа индексатора) для указанного объекта. В качестве неявного параметра value методу будет передана строка "Hello World".

Индексаторы и присваивание

В примере 9.9 запрещено присваивать что-либо элементам массивов, не имеющим никакого значения. То есть если написать:

```
lbt[10] = "wow!";
```

будет вызван обработчик ошибки из процедуры доступа set(), который уведомит программиста, что переданное значение индекса (10) больше значения счетчика (6).

Конечно, процедурой доступа set() можно пользоваться для присваивания, необходимо лишь проверять передаваемые методу индексы. С этой целью можно переписать процедуру доступа set() так, чтобы она проверяла свойство Length буфера, а не значение счетчика. Если будет передано значение для элемента, еще не имеющего значения, можно будет просто скорректировать счетчик:

```
set
{
   // добавление только с помощью метода add
   if (index >= strings.Length )
   {
      // обработка ошибки
   }
   else
   {
      strings[index] = value;
      if (ctr < index+1)
         ctr = index+1;
   }
}
```

Так можно будет создавать разреженный массив, в котором разрешается присвоить значение элементу с индексом 10, даже если у элемента с индексом 9 значения еще нет. То есть если написать:

```
lbt[10] = "wow!";
```

то программа выведет:

```
lbt[0]: Hello
lbt[1]: Universe
lbt[2]: Who
```

```
lbt[3]: Is
lbt[4]: John
lbt[5]: Galt
lbt[6]:
lbt[7]:
lbt[8]:
lbt[9]:
lbt[10]: wow!
```

В методе Main() создается экземпляр класса ListBoxTest по имени lbt, причем конструктору в качестве аргументов передаются две строки:

```
ListBoxTest lbt = new ListBoxTest("Hello", "World");
```

Затем вызывается метод Add(), добавляющий еще четыре:

```
// добавление в список нескольких строк
lbt.Add("Who");
lbt.Add("Is");
lbt.Add("John");
lbt.Add("Galt");
```

Перед выводом результатов на консоль второе значение (с индексом 1) изменяется:

```
string subst = "Universe";
lbt[1] = subst;
```

Наконец, в цикле выводится каждое значение:

```
for (int i = 0;i<lbt.GetNumEntries();i++)
{
    Console.WriteLine("lbt[{0}]: {1}",i,lbt[i]);
}
```

Индексы других типов

C# не требует, чтобы индексом объекта класса коллекции обязательно было целое число. Создавая свой класс коллекции и индексатор, программист может использовать индексы строкового или любого другого типа. Индексатор может быть перегружен в данном классе коллекции таким образом, что в зависимости от требований пользователя для доступа может быть использован, например, как строковый, так и целочисленный индекс.

При работе с окном списка строковый индекс, возможно, даже предпочтительнее. В примере 9.10 иллюстрируется применение строки в качестве индекса. Индексатор вызывает вспомогательный метод findString(), который возвращает индекс строки, переданной ему в качестве аргумента. Обратите внимание, что перегруженный индексатор мирно сосуществует с индексатором из примера 9.9.

Пример 9.10. Перегрузка индексатора

```
namespace Programming_CSharp
{
   using System;

   // упрощенный элемент управления ListBox
   public class ListBoxTest
   {
      // инициализация списка строками
      public ListBoxTest(params string[] initialStrings)
      {
         // выделение памяти под строки
         strings = new String[256];

         // копирование строк, переданных конструктору
         foreach (string s in initialStrings)
         {
            strings[ctr++] = s;
         }
      }

      // добавление одиночной строки в конец списка
      public void Add(string theString)
      {
         strings[ctr] = theString;
         ctr++;
      }

      // реализация доступа "в стиле массива"
      public string this[int index]
      {
         get
         {
            if (index < 0 || index >= strings.Length)
            {
               // обработка недопустимого индекса
            }
            return strings[index];
         }
         set
         {
            strings[index] = value;
         }
      }

      private int findString(string searchString)
      {
         for (int i = 0;i<strings.Length;i++)
         {
            if (strings[i].StartsWith(searchString))
            {
               return i;
            }
```

```
        }
        return -1;
    }
    // индексация строкой
    public string this[string index]
    {
        get
        {
            if (index.Length == 0)
            {
                // обработка недопустимого индекса
            }

            return this[findString(index)];
        }
        set
        {
            strings[findString(index)] = value;
        }
    }

    // возвращение количества строк в списке
    public int GetNumEntries()
    {
        return ctr;
    }

    private string[] strings;
    private int ctr = 0;
}

public class Tester
{
    static void Main()
    {
        // создание и инициализация нового списка
        ListBoxTest lbt =
            new ListBoxTest("Hello", "World");

        // добавление в список нескольких строк
        lbt.Add("Who");
        lbt.Add("Is");
        lbt.Add("John");
        lbt.Add("Galt");

        // тестирование доступа
        string subst = "Universe";
        lbt[1] = subst;
        lbt["Hel"] = "GoodBye";
        // lbt["xyz"] = "oops";

        // обращение ко всем строкам
        for (int i = 0;i<lbt.GetNumEntries();i++)
        {
            Console.WriteLine("lbt[{0}]: {1}",i,lbt[i]);
```

```
        }       // конец for
    }           // конец main
  }             // конец tester
}               // конец namespace
```

Вывод:

```
lbt[0]: GoodBye
lbt[1]: Universe
lbt[2]: Who
lbt[3]: Is
lbt[4]: John
lbt[5]: Galt
```

Пример 9.10 практически идентичен примеру 9.9, если не считать появления перегруженного индексатора, который, если потребуется, может выполнить поиск строки, и метода `findString()`, обеспечивающего этот поиск.

Метод `findString()` просто перебирает в цикле все строки массива `strings`, пока не найдет такую, которая начинается со строки, переданной в качестве индекса. Если поиск закончился успешно, возвращается индекс найденной строки, в противном случае возвращается `-1`.

В методе `Main()` индексатору передается фрагмент строки, аналогично тому, как до сих пор передавалось целое:

```
lbt["Hel"] = "GoodBye";
```

Вызывается перегруженный индексатор, выполняющий минимальную проверку (в данном случае необходимо убедиться, что переданная строка содержит хотя бы один символ) и передающий значение (`Hel`) методу `findString()`. Значение, возвращенное методом, индексатор использует для обращения к элементу массива `strings`:

```
return this[findString(index)];
```

Аналогично работает процедура доступа `set()`:

```
strings[findString(index)] = value;
```

Внимательные читатели, конечно, заметили, что при неуспешном окончании поиска возвращается значение `-1`, которое затем используется в качестве индекса массива `strings`. Это действие приводит к возникновению исключения (`System.NullReferenceException`), в чем нетрудно убедиться, сняв комментарий со строки:

```
lbt["xyz"] = "oops";
```

в методе `Main()`. Корректная обработка случая отсутствия в списке искомой строки предлагается читателю в качестве самостоятельного упражнения. Можно выдать сообщение об ошибке либо как-то иначе позволить пользователю справиться с ситуацией.

Интерфейсы классов коллекций

Платформа .NET Framework предоставляет программистам ряд стандартных интерфейсов для создания, перечисления и сравнения объектов классов коллекций. Основные из них приведены в табл. 9.2.

Таблица 9.2. Интерфейсы классов коллекций

Интерфейс	Назначение
IEnumerable	Составляет список объектов классов коллекций с помощью оператора foreach
ICollection	Реализуется всеми классами коллекций для обеспечения доступа к методу CopyTo() и свойствам Count, IsSynchronized и SyncRoot
IComparer	Сравнивает два объекта классов коллекций при их сортировке
IList	Используется объектами классов коллекций, индексируемыми как массив
IDictionary	Используется классами коллекций, осуществляющими доступ по ключу или значению, таких как Hashtable и SortedList
IdictionaryEnumerator	Позволяет просмотреть (с помощью оператора foreach) объекты классов коллекций, поддерживающих интерфейс IDictionary

Интерфейс IEnumerable

Чтобы класс ListBoxTest поддерживал оператор foreach, можно реализовать в нем интерфейс IEnumerable. У этого интерфейса только один метод, GetEnumerator(), чья задача – возвращать специализированную реализацию интерфейса IEnumerator. Иными словами, семантика класса Enumerable такова, что он предоставляет класс Enumerator:

```
public IEnumerator GetEnumerator()
{
    return (IEnumerator) new ListBoxEnumerator(this);
}
```

Класс Enumerator должен реализовать методы и свойства интерфейса IEnumerator. Они могут быть реализованы либо непосредственно классом контейнера (в данном случае классом ListBoxTest), либо отдельным классом. Последний подход, вообще говоря, предпочтительнее, поскольку он инкапсулирует всю ответственность в классе Enumerator, не загромождая контейнер.

Поскольку класс Enumerator специфичен для класса контейнера (то есть, например, ListBoxEnumerator должен «знать все» о классе ListBoxTest), реализация должна быть закрытой и внутренней для класса ListBoxTest.

Обратите внимание, что метод передает перечислителю текущий объект класса ListBoxTest (обозначив его ключевым словом this), что позволяет ему перечислять элементы этого конкретного объекта ListBoxTest.

Класс, реализующий перечисление, представлен здесь как ListBoxEnumerator – закрытый класс, определенный *внутри* класса ListBoxTest. Его логика проста. Он должен реализовать в объекте открытое свойство Current и два открытых метода, MoveNext() и Reset().

Перечисляемый класс ListBoxTest передается конструктору в качестве аргумента. Там он присваивается переменной lbt. Кроме того, конструктор присваивает переменной index значение -1, которое является индикатором того, что перечисление элементов еще не началось:

```
public ListBoxEnumerator(ListBoxTest lbt)
{
   this.lbt = lbt;
   index = -1;
}
```

Метод MoveNext() инкрементирует индекс и затем проверяет, не вышел ли он за пределы перечисляемого объекта. Если это так, возвращается значение false, в противном случае – true:

```
public bool MoveNext()
{
   index++;
   if (index >= lbt.strings.Length)
      return false;
   else
      return true;
}
```

Метод Reset() класса ListBoxEnumerator всего лишь сбрасывает индекс в значение -1.

Свойство Current реализуется так, что возвращает текущую строку. Это совершенно произвольное решение; в других классах свойство Current будет иметь такой смысл, какой придаст ему разработчик. Как бы то ни было, каждый перечислитель должен быть в состоянии возвращать текущий элемент, поскольку предоставление доступа к текущему элементу – его непосредственная задача:

```
public object Current
{
   get
   {
      return(lbt[index]);
   }
}
```

Ну, вот и все. Обращение к оператору foreach приводит к вызову перечислителя, который перебирает элементы массива. Поскольку опера-

тор foreach выводит на экран все строки, независимо от того, было ли им присвоено значение, измените определение массива strings так, чтобы он содержал только восемь элементов (пример 9.11). В противном случае читать вывод программы будет неудобно.

Пример 9.11. Превращение класса ListBox в перечислитель

```
namespace Programming_CSharp
{
   using System;
   using System.Collections;

   // упрощенный элемент управления ListBox
   public class ListBoxTest : IEnumerable
   {
      // закрытая реализация класса ListBoxEnumerator
      private class ListBoxEnumerator : IEnumerator
      {
         // открытый в рамках закрытой реализации;
         // следовательно, закрытый в рамках класса ListBoxTest
         public ListBoxEnumerator(ListBoxTest lbt)
         {
            this.lbt = lbt;
            index = -1;
         }

         // инкрементировать индекс и убедиться,
         // что значение допустимо
         public bool MoveNext()
         {
            index++;
            if (index >= lbt.strings.Length)
               return false;
            else
               return true;
         }

         public void Reset()
         {
            index = -1;
         }
         // свойство Current определено как строка,
         // добавленная в список последней
         public object Current
         {
            get
            {
               return(lbt[index]);
            }
         }

         private ListBoxTest lbt;
         private int index;
      }
```

```
    // перечисляемые классы могут возвращать перечислитель
    public IEnumerator GetEnumerator()
    {
       return (IEnumerator) new ListBoxEnumerator(this);
    }

    // инициализация списка строками
    public ListBoxTest(params string[] initialStrings)
    {
       // выделение памяти под строки
       strings = new String[8];

       // копирование строк, переданных конструктору
       foreach (string s in initialStrings)
       {
          strings[ctr++] = s;
       }
    }

    // добавление одиночной строки в конец списка
    public void Add(string theString)
    {
       strings[ctr] = theString;
       ctr++;
    }

    // реализация доступа как к массиву
    public string this[int index]
    {
       get
       {
          if (index < 0 || index >= strings.Length)
          {
             // обработка недопустимого индекса
          }
          return strings[index];
       }
       set
       {
          strings[index] = value;
       }
    }

    // возвращение числа строк в списке
    public int GetNumEntries()
    {
       return ctr;
    }

    private string[] strings;
    private int ctr = 0;
}
public class Tester
{
```

```
        static void Main()
        {
           // создание и инициализация нового списка
           ListBoxTest lbt =
              new ListBoxTest("Hello", "World");

           // добавление в список нескольких строк
           lbt.Add("Who");
           lbt.Add("Is");
           lbt.Add("John");
           lbt.Add("Galt");

           // тестирование доступа
           string subst = "Universe";
           lbt[1] = subst;

           // обращение ко всем строкам
           foreach (string s in lbt)
           {
              Console.WriteLine("Значение: {0}", s);
           }
        }
     }
  }
```

Вывод:

```
Значение: Hello
Значение: Universe
Значение: Who
Значение: Is
Значение: John
Значение: Galt
Значение:
Значение:
```

Выполнение программы начинается в методе `Main()` с создания нового объекта типа `ListBoxTest` и передачи конструктору двух строк. В момент создания объекта создается массив `strings` из восьми элементов. Еще четыре строки добавляются с помощью метода `Add()`, а вторая строка массива изменяется, совсем как в предыдущем примере.

Существенным отличием этой версии программы от предыдущей является наличие цикла `foreach`, который просматривает все строки в списке. Оператор `foreach` автоматически использует интерфейс `IEnumerable`, вызывая метод `GetEnumerator()`. Метод возвращает объект `ListBoxEnumerator`, для которого вызывается конструктор, инициализирующий индекс значением `-1`.

Затем оператор `foreach` вызывает метод `MoveNext()`, который инкрементирует индекс до 0 и возвращает `true`. После этого `foreach` обращается к свойству `Current` и получает текущую строку. Само свойство `Current` вызывает индексатор списка, тем самым получая строку, имеющую индекс `0`. Эта строка присваивается переменной `s`, определенной в за-

головке оператора foreach, и s выводится на консоль. Оператор foreach повторяет описанные шаги (вызов MoveNext(), обращение к Current, вывод на экран), пока не переберет все строки в списке.

Интерфейс ICollection

Еще одним важным интерфейсом, предоставляемым платформой .NET Framework для работы с массивами и классами коллекций, является ICollection. Он обладает тремя свойствами: Count, IsSynchronized и SyncRoot, а также одним методом CopyTo(). Метод будет рассмотрен далее в этой главе, а из свойств чаще других используется Count, которое возвращает количество элементов в коллекции:

```
for (int i = 0;i<myIntArray.Count;i++)
{
   //...
}
```

Здесь свойство Count массива myIntArray используется для выяснения числа элементов массива, например при выводе их значений.

Интерфейсы IComparer и IComparable

Интерфейс IComparer содержит метод Compare(), с помощью которого можно сравнить два элемента коллекции. Можно реализовать IComparer во вспомогательном классе, который передается перегруженным методам, таким как Array.Sort(Array a, IComparer c). Интерфейс IComparable выполняет схожую роль, но он описывает метод Compare() для сравниваемого объекта, а не во вспомогательном классе.

Как правило, реализация метода Compare() включает в себя вызов метода CompareTo() для одного из объектов. Метод CompareTo() имеется у всех объектов, реализующих интерфейс IComparable. Если программист хочет создать класс, который можно отсортировать внутри коллекции, он должен будет реализовать интерфейс IComparable.

Платформа .NET Framework предоставляет класс Comparer, реализующий интерфейс IComparer, причем его реализация по умолчанию учитывает регистр символов. В следующем разделе показано, как создать собственную реализацию интерфейсов IComparable и IComparer.

Класс ArrayList

Классической проблемой, связанной с типом Array, является его фиксированный размер. Если программист заранее не знает, сколько объектов будет содержать массив, он рискует объявить либо слишком маленький массив (в котором окажется недостаточно места), либо слишком большой (нерационально расходующий память).

Нередко программа запрашивает информацию у пользователя или получает ее на веб-сайте. По мере поступления объектов (строк, книг, числовых значений и т. д.) программа записывает их в массив, однако количество объектов, собираемых за один сеанс, предугадать невозможно. Классический массив фиксированного размера не годится, поскольку никто не скажет, какой размер потребуется.

Класс ArrayList является массивом, чей размер динамически увеличивается по мере необходимости. Объекты ArrayList обладают рядом полезных свойств и методов, позволяющих эффективно работать с ними. Наиболее важные из этих свойств и методов приведены в табл. 9.3.

Таблица 9.3. Свойства и методы класса ArrayList

Метод или свойство	Предназначение
Adapter()	Открытый статический метод, создающий оболочку ArrayList для объекта IList
FixedSize()	Перегруженный открытый статический метод, возвращающий объект-список в качестве оболочки. Этот список имеет фиксированный размер; его элементы можно изменять, но нельзя удалять или добавлять новые
ReadOnly()	Перегруженный открытый статический метод, возвращающий в качестве оболочки класс списка, предоставляющий доступ только для чтения
Repeat()	Открытый статический метод, возвращающий объект ArrayList, элементы которого являются копиями указанного значения
Synchronized()	Перегруженный открытый статический метод, возвращающий оболочку, безопасную с точки зрения многопоточности
Capacity	Свойство, позволяющее считывать и изменять количество элементов, которое может содержать объект ArrayList
Count	Свойство, используемое только для чтения, позволяющее узнать текущее количество элементов массива
IsFixedSize	Свойство, доступное только для чтения, позволяющее выяснить, имеет ли объект ArrayList фиксированный размер
IsReadOnly	Свойство, позволяющее выяснить, доступен ли объект ArrayList только для чтения
IsSynchronized	Свойство, доступное только для чтения, позволяющее выяснить, обеспечивает ли объект ArrayList безопасную работу с несколькими потоками
Item()	Считывает или изменяет значение элемента, имеющего указанный индекс. Фактически является индексатором класса ArrayList
SyncRoot	Открытое свойство, возвращающее объект, с помощью которого можно синхронизировать доступ к объекту ArrayList
Add()	Открытый метод, добавляющий объект в массив ArrayList

Метод или свойство	Предназначение
AddRange()	Открытый метод, добавляющий элементы ICollection в конец массива ArrayList
BinarySearch()	Перегруженный открытый метод, который при помощи двоичного поиска находит указанный элемент в отсортированном массиве ArrayList
Clear()	Удаляет все элементы из объекта ArrayList
Clone()	Создает поверхностную копию объекта
Contains()	Выясняет, находится ли данный элемент в массиве ArrayList
CopyTo()	Перегруженный открытый метод, копирующий объект ArrayList в одномерный массив
GetEnumerator()	Перегруженный открытый метод, возвращающий перечислитель для перебора элементов объекта ArrayList
GetRange()	Копирует последовательность элементов в новый объект ArrayList
IndexOf()	Перегруженный открытый метод, возвращающий индекс первого появления указанного значения
Insert()	Вставляет элемент в объект ArrayList
InsertRange()	Вставляет элементы коллекции в объект ArrayList
LastIndexOf()	Перегруженный открытый метод, возвращающий индекс последнего появления указанного значения в объекте ArrayList
Remove()	Удаляет первое вхождение указанного объекта
RemoveAt()	Удаляет элемент, имеющий указанный индекс
RemoveRange()	Удаляет последовательность элементов
Reverse()	Меняет порядок следования элементов объекта ArrayList на обратный
SetRange()	Копирует элементы коллекции в указанный диапазон элементов массива ArrayList
Sort()	Сортирует ArrayList
ToArray()	Копирует элементы объекта ArrayList в новый массив
TrimToSize()	Присваивает свойству Capacity значение, равное фактическому количеству элементов в объекте ArrayList

Создавая объект ArrayList, программист не указывает количество элементов, которое он будет содержать. Новые элементы добавляются с помощью метода Add(), а объект сам ведет их учет, что подтверждается примером 9.12.

Пример 9.12. Работа с классом ArrayList

```
namespace Programming_CSharp
{
```

```
using System;
using System.Collections;

// простой класс для хранения в массиве
public class Employee
{
   public Employee(int empID)
   {
      this.empID = empID;
   }
   public override  string ToString()
   {
      return empID.ToString();
   }
   public int EmpID
   {
      get
      {
         return empID;
      }
      set
      {
         empID = value;
      }
   }

   private int empID;
}
public class Tester
{
   static void Main()
   {
      ArrayList empArray = new ArrayList();
      ArrayList intArray = new ArrayList();

      // заполнение массива
      for (int i = 0;i<5;i++)
      {
         empArray.Add(new Employee(i+100));
         intArray.Add(i*5);
      }

      // вывод содержимого массива
      for (int i = 0;i<intArray.Count;i++)
      {
         Console.Write("{0} ", intArray[i].ToString());
      }

      Console.WriteLine("\n");

      // вывод содержимого массива empArray
      for (int i = 0;i<empArray.Count;i++)
      {
         Console.Write("{0} ", empArray[i].ToString());
```

```
            }

            Console.WriteLine("\n");
            Console.WriteLine("empArray.Capacity: {0}",
               empArray.Capacity);
        }
    }
}
```

Вывод:

```
0 5 10 15 20
100 101 102 103 104
empArray.Capacity: 16
```

Для класса Array программист сразу определяет, сколько элементов будет в массиве. При попытке записать в массив больше элементов, чем определено, класс Array вызовет исключение. Что касается класса ArrayList, количество его элементов не объявляется. Класс ArrayList имеет свойство Capacity, сообщающее, сколько элементов способен вместить объект ArrayList:

```
public int Capacity {virtual get; virtual set; }
```

По умолчанию число элементов массива равно 16. При добавлении семнадцатого элемента свойство Capacity автоматически удваивается до 32. Так, если изменить цикл for на:

```
for (int i = 0;i<17;i++)
```

то вывод будет выглядеть следующим образом:

```
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
empArray.Capacity: 32
```

Программист может вручную присвоить свойству Capacity любое значение, которое больше или равно значению свойства Count. Если попытаться присвоить свойству Capacity значение, меньшее Count, программа вызовет исключение ArgumentOutOfRangeException.

Реализация интерфейса IComparable

Как и все классы коллекций, класс ArrayList реализует метод Sort(), позволяющий сортировать объекты, реализующие интерфейсы IComparable. В следующем примере объект Employee будет изменен таким образом, чтобы он реализовал интерфейс IComparable:

```
public class Employee : IComparable
```

Чтобы реализовать этот интерфейс, объект Employee должен реализовать метод CompareTo():

```
public int CompareTo(Object rhs)
{
```

```
        Employee r = (Employee) rhs;
        return this.empID.CompareTo(r.empID);
    }
```

Метод CompareTo() принимает в качестве аргумента некоторый объект. Объект Employee сравнивает себя с этим объектом и возвращает -1, если он меньше объекта, 1, если он больше того объекта, и 0, если оба объекта равны. Объект Employee сам определяет, какой смысл кроется за словами «больше», «меньше» и «равно». Например, можно провести сравниваемый объект к типу Employee, а затем делегировать сравнение элементу empId, который имеет тип int и вызывает метод CompareTo(), по умолчанию сравнивающий два целых числа.

Поскольку тип int является потомком класса Object, он обладает методами, в том числе методом CompareTo(). Так, значение типа int становится объектом, которому можно делегировать ответственность за выполнение сравнения.

Теперь все готово для сортировки empList, списка объектов Employee. Чтобы убедиться, что сортировка работает, необходимо заполнить случайными числами целочисленный массив и массив объектов класса Employee. Для получения случайных значений создадим объект класса Random. Если теперь вызвать метод Next() объекта Random, он возвратит псевдослучайное число. Так можно создать последовательность случайных чисел. Метод Next() является перегруженным; одна из версий принимает в качестве аргумента целое число, уточняющее максимально допустимое случайное число. В данном примере методу Next() передается 10, чтобы он генерировал случайное число от 0 до 10:

```
Random r = new Random();
r.Next(10);
```

В примере 9.13 создаются целочисленный массив и массив объектов типа Employee. Оба массива заполняются случайными значениями, которые тут же выводятся на экран. Затем массивы сортируются и опять выводятся.

Пример 9.13. Сортировка массива целых и массива объектов Employee

```
namespace Programming_CSharp
{
    using System;
    using System.Collections;

    // простой класс для хранения в массиве
    public class Employee : IComparable
    {
        public Employee(int empID)
        {
            this.empID = empID;
```

```
    }

    public override  string ToString()
    {
       return empID.ToString();
    }

    // объект Comparer делегирует сравнение объекту Employee,
    // который пользуется методом CompareTo(),
    // установленным по умолчанию для целых чисел
    public int CompareTo(Object rhs)
    {
       Employee r = (Employee) rhs;
       return this.empID.CompareTo(r.empID);
    }

    private int empID;
 }
 public class Tester
 {
    static void Main()
    {
       ArrayList empArray = new ArrayList();
       ArrayList intArray = new ArrayList();

       // генерирование случайных чисел
       // для массива целых и
       // массива объектов Employee
       Random r = new Random();

       // заполнение массива
       for (int i = 0;i<5;i++)
       {
          // добавление случайного значения для объекта Employee
          empArray.Add(new Employee(r.Next(10)+100));

          // добавление случайного целого числа
          intArray.Add(r.Next(10));
       }

       // вывод содержимого целочисленного массива
       for (int i = 0;i<intArray.Count;i++)
       {
          Console.Write("{0} ", intArray[i].ToString());
       }
       Console.WriteLine("\n");
       // вывод содержимого массива объектов Employee
       for (int i = 0;i<empArray.Count;i++)
       {
          Console.Write("{0} ", empArray[i].ToString());
       }
       Console.WriteLine("\n");

       // сортировка и вывод содержимого целочисленного массива
```

```
            intArray.Sort();
            for (int i = 0;i<intArray.Count;i++)
            {
               Console.Write("{0} ", intArray[i].ToString());
            }
            Console.WriteLine("\n");

            // сортировка и вывод содержимого массива объектов Employee
            //Employee.EmployeeComparer c = Employee.GetComparer();
            //empArray.Sort(c);
            empArray.Sort();

            // вывод содержимого массива объектов Employee
            for (int i = 0;i<empArray.Count;i++)
            {
               Console.Write("{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");

         }
      }
   }
```

Вывод:

```
8 5 7 3 3
105 103 102 104 106
3 3 5 7 8
102 103 104 105 106
```

Из выведенного данной программой текста понятно, что оба массива вначале были заполнены случайными числами, а затем отсортированы, и их элементы теперь следуют в правильном порядке.

Реализация интерфейса IComparer

Когда вызывается метод `Sort()` объекта `ArrayList`, вызывается предоставляемая по умолчанию реализация интерфейса `IComparer`. В ней применяется метод быстрой сортировки (`QuickSort`), вызывающий для каждого элемента объекта `ArrayList` метод `CompareTo()` из реализации интерфейса `IComparable`.

Программист может создать собственную реализацию интерфейса `IComparer`, которая удовлетворит конкретным требованиям, предъявляемым к процессу сортировки. Например, в следующем коде в класс `Employee` добавлено второе поле, `yearsOfSvc`. Требуется, чтобы объекты `Employee` из массива `ArrayList` можно было сортировать по любому из полей, `empID` или `yearsOfSvc`.

Чтобы добиться этого, приходится создать собственную реализацию интерфейса `IComparer`, которая будет передана методу `Sort()` объекта `ArrayList`. Класс `EmployeeComparer`, реализующий интерфейс `IComparer`, знает, что такое объекты `Employee` и как их сортировать.

В классе EmployeeComparer реализовано свойство WhichComparison, имеющее тип Employee.EmployeeComparer.ComparisonType:

```
public Employee.EmployeeComparer.ComparisonType
   WhichComparison
{
   get
   {
      return whichComparison;
   }
   set
   {
      whichComparison=value;
   }
}
```

Здесь ComparisonType является перечислением, имеющим два значения и определяющим, по какому из полей (empID или yearsOfSvc) следует проводить сортировку:

```
public enum ComparisonType
{
   EmpID,
   Yrs
};
```

Прежде чем вызвать метод Sort(), нужно создать объект класса EmployeeComparer и установить его свойство ComparisonType:

```
Employee.EmployeeComparer c = Employee.GetComparer();
c.WhichComparison=Employee.EmployeeComparer.ComparisonType.EmpID;
empArray.Sort(c);
```

Теперь вызов метода Sort() для объекта ArrayList повлечет за собой вызов метода Compare() объекта EmployeeComparer. Тот, в свою очередь, делегирует сравнение методу Employee.CompareTo(), передав ему свойство WhichComparison:

```
public int Compare(object lhs, object rhs)
{
   Employee l = (Employee) lhs;
   Employee r = (Employee) rhs;
   return l.CompareTo(r,WhichComparison);
}
```

Объект Employee обязан реализовать собственную версию метода CompareTo(), которая сравнивала бы объекты должным образом:

```
public int CompareTo(
    Employee rhs,
    Employee.EmployeeComparer.ComparisonType which)
{
```

```
        switch (which)
        {
            case Employee.EmployeeComparer.ComparisonType.EmpID:
                return this.empID.CompareTo(rhs.empID);
            case Employee.EmployeeComparer.ComparisonType.Yrs:
                return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);
        }
        return 0;
    }
```

Полный текст программы рассматриваемого случая приводится в примере 9.14. Массив целых чисел убран ради упрощения программы, зато вывод информации доработан так, что результаты сортировки предстают максимально наглядно.

Пример 9.14. Сортировка массива сотрудников по табельным номерам (поле empID) и по стажу работы в фирме (поле yearsOfSvc)

```
namespace Programming_CSharp
{
    using System;
    using System.Collections;

    // простой класс для хранения в массиве
    public class Employee : IComparable
    {
        public Employee(int empID)
        {
            this.empID = empID;
        }

        public Employee(int empID, int yearsOfSvc)
        {
            this.empID = empID;
            this.yearsOfSvc = yearsOfSvc;
        }

        public override  string ToString()
        {
            return "ID: " + empID.ToString() +
            ". Years of Svc: " + yearsOfSvc.ToString();
        }

        // статический метод для получения объекта Comparer
        public static EmployeeComparer GetComparer()
        {
            return new Employee.EmployeeComparer();
        }

        // объект Comparer делегирует сравнение объекту Employee,
        // который пользуется методом CompareTo(),
        // установленным по умолчанию для целых чисел
        public int CompareTo(Object rhs)
        {
```

```
      Employee r = (Employee) rhs;
      return this.empID.CompareTo(r.empID);
   }

   // специальная реализация, вызываемая собственным объектом Comparer
   public int CompareTo(
      Employee rhs,
      Employee.EmployeeComparer.ComparisonType which)
   {
      switch (which)
      {
         case Employee.EmployeeComparer.ComparisonType.EmpID:
            return this.empID.CompareTo(rhs.empID);
         case Employee.EmployeeComparer.ComparisonType.Yrs:
            return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);
      }
      return 0;

   }

   // вложенный класс, реализующий интерфейс IComparer
   public class EmployeeComparer : IComparer
   {
      // перечисление, содержащее виды сравнения
      public enum ComparisonType
      {
         EmpID,
         Yrs
      };

      // сообщает объектам Employee о необходимости сравнить свои значения
      public int Compare(object lhs, object rhs)
      {
         Employee l = (Employee) lhs;
         Employee r = (Employee) rhs;
         return l.CompareTo(r,WhichComparison);
      }

      public Employee.EmployeeComparer.ComparisonType
         WhichComparison
      {
         get
         {
            return whichComparison;
         }
         set
         {
            whichComparison=value;
         }
      }

      // закрытая переменная состояния
      private Employee.EmployeeComparer.ComparisonType
         whichComparison;
   }
```

```
        private int empID;
        private int yearsOfSvc = 1;
    }
    public class Tester
    {
        static void Main()
        {
            ArrayList empArray = new ArrayList();

            // генерирование случайных чисел
            // для массива целых и
            // массива объектов Employee
            Random r = new Random();

            // заполнение массива
            for (int i = 0;i<5;i++)
            {
                // добавление случайного значения для объекта Employee
                empArray.Add(
                    new Employee(
                        r.Next(10)+100,r.Next(20)
                    )
                );
            }

            // вывод содержимого массива объектов Employee
            for (int i = 0;i<empArray.Count;i++)
            {
                Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");

            // сортировка и вывод содержимого массива объектов Employee
            Employee.EmployeeComparer c = Employee.GetComparer();
            c.WhichComparison=Employee.EmployeeComparer.ComparisonType.EmpID;
            empArray.Sort(c);
            // вывод содержимого массива объектов Employee
            for (int i = 0;i<empArray.Count;i++)
            {
                Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");
            c.WhichComparison=Employee.EmployeeComparer.ComparisonType.Yrs;
            empArray.Sort(c);
            for (int i = 0;i<empArray.Count;i++)
            {
                Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");

        }
    }
}
```

Вывод:

```
ID: 103. Years of Svc: 11
ID: 108. Years of Svc: 15
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 5
ID: 102. Years of Svc: 0

ID: 102. Years of Svc: 0
ID: 103. Years of Svc: 11
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 15
ID: 108. Years of Svc: 5

ID: 102. Years of Svc: 0
ID: 108. Years of Svc: 5
ID: 103. Years of Svc: 11
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 15
```

Первый блок выводимых значений демонстрирует объекты `Employee` в порядке их добавления в массив. Значения обоих полей случайны.

Во втором блоке представлены результаты сортировки по полю `empID`, а в третьем – по полю `yearsOfSvc`.

Очереди

Очередь (*queue*) – это класс коллекции, организованный по принципу FIFO (первым вошел – первым вышел). Классическая аналогия – очередь в кассу за билетами. Первый человек, стоящий в очереди, первым и выйдет из нее, когда купит билет.

Очередь удачно подходит для управления ограниченными ресурсами. Пусть, например, некий ресурс принимает сообщения и обрабатывает их по одному. Программист создает очередь сообщений, говоря клиентам: «Ваше сообщение очень важно для нас. Все сообщения обрабатываются в порядке поступления».

Класс `Queue` обладает рядом методов и свойств, перечисленных в табл. 9.4.

Таблица 9.4. Свойства и методы класса Queue

Метод или свойство	Предназначение
`Synchronized()`	Открытый статический метод, возвращающий оболочку типа `Queue`, обеспечивающую безопасную работу с несколькими потоками
`Count`	Открытое свойство, позволяющее узнать текущее количество элементов очереди
`IsSynchronized`	Открытое свойство, позволяющее выяснить, обеспечивает ли объект `Queue` безопасную работу с несколькими потоками

Таблица 9.4 (продолжение)

Метод или свойство	Предназначение
SyncRoot	Открытое свойство, возвращающее объект, с помощью которого можно синхронизировать доступ к объекту Queue
Clear()	Удаляет все элементы из объекта Queue
Clone()	Создает копию объекта
Contains()	Выясняет, находится ли данный элемент в объекте Queue
CopyTo()	Копирует объект Queue в существующий одномерный массив
Dequeue()	Возвращает объект, стоящий в начале объекта Queue, и удаляет его из очереди
Enqueue()	Добавляет объект в конец объекта Queue
GetEnumerator()	Возвращает перечислитель объекта Queue
Peek()	Возвращает объект, стоящий в начале объекта Queue, не удаляя его
ToArray()	Копирует элементы объекта Queue в новый массив

Новые элементы добавляются в очередь методом Enqueue(), а удаляются из нее методом Dequeue() или при помощи перечислителя. Иллюстрация – в примере 9.15.

Пример 9.15. Работа с очередями

```
namespace Programming_CSharp
{
   using System;
   using System.Collections;

   public class Tester
   {

      static void Main()
      {
         Queue intQueuee = new Queue();

         // заполнить очередь
         for (int i = 0;i<5;i++)
         {
            intQueuee.Enqueue(i*5);
         }

         // вывести объект Queue
         Console.Write( "Значения intQueuee:\t" );
         PrintValues( intQueuee );

         // удалить элемент очереди
         Console.WriteLine(
            "\n(Dequeue)\t{0}", intQueuee.Dequeue() );
```

```
            // вывести объект Queue
            Console.Write( "Значения intQueuee:\t" );
            PrintValues( intQueuee );

            // удалить еще один элемент из очереди
            Console.WriteLine(
               "\n(Dequeue)\t{0}", intQueuee.Dequeue() );

            // вывести объект Queue
            Console.Write( "Значения intQueuee:\t" );
            PrintValues( intQueuee );

            // прочитать первый элемент из очереди,
            // не удаляя его
            Console.WriteLine(
               "\n(Peek)   \t{0}", intQueuee.Peek() );

            // вывести объект Queue
            Console.Write( "Значения intQueuee:\t" );
            PrintValues( intQueuee );

         }

         public static void PrintValues( IEnumerable myCollection )
         {
            IEnumerator myEnumerator =
               myCollection.GetEnumerator();
            while ( myEnumerator.MoveNext() )
               Console.Write( "{0} ",myEnumerator.Current );
            Console.WriteLine();
         }

      }
   }
```

Вывод:

```
Значения intQueuee:       0 5 10 15 20
(Dequeue)       0
Значения intQueuee:       5 10 15 20

(Dequeue)       5
Значения intQueuee:       10 15 20

(Peek)          10
Значения intQueuee:       10 15 20
```

В этом примере объект `ArrayList` заменен на объект `Queue`. От класса `Employee` пришлось отказаться ради экономии места, но, конечно, ставить в очередь можно и объекты, имеющие тип, определенный пользователем.

Выводимые программой данные показывают, что метод `Enqueue()` добавляет элементы в объект `Queue`, а метод `Dequeue()` возвращает их, одновременно удаляя из очереди. Метод `Peek()` объекта `Queue` позволяет прочитать первый элемент очереди, не удаляя его.

Поскольку класс Queue является перечисляемым, его можно передать методу PrintValues() в качестве интерфейса IEnumerable. Преобразование происходит неявно. Метод PrintValues() вызывает метод GetEnumerator(), который, как помнит читатель, является единственным методом всех классов IEnumerable. Метод GetEnumerator() возвращает объект Enumerator, используемый для перечисления объектов коллекции.

Стек

Стек (*stack*) – это класс коллекции, организованный по принципу LIFO (последним вошел – первым вышел). Аналогией ему может служить стопка подносов в столовой самообслуживания (или, например, столбик из монет). Поднос, положенный в стопку последним, будет взят оттуда первым.

Основными методами для работы со стеком являются Push() (добавление элемента) и Pop() (удаление). Кроме того, класс Stack предоставляет метод Peek(), аналогичный одноименному методу класса Queue. Важнейшие методы и свойства класса Stack приведены в табл. 9.5.

Таблица 9.5. Методы и свойства класса Stack

Метод или свойство	Предназначение
Synchronized()	Открытый статический метод, возвращающий оболочку типа Stack, обеспечивающую безопасную работу с несколькими потоками
Count	Открытое свойство, позволяющее узнать текущее количество элементов стека
IsSynchronized	Открытое свойство, позволяющее выяснить, обеспечивает ли объект Stack безопасность при работе с несколькими потоками
SyncRoot	Открытое свойство, возвращающее объект, с помощью которого можно синхронизировать доступ к объекту Stack
Clear()	Удаляет все элементы из объекта Stack
Clone()	Создает копию объекта
Contains()	Выясняет, находится ли данный элемент в объекте Stack
CopyTo()	Копирует объект Stack в существующий одномерный массив
GetEnumerator()	Возвращает перечислитель для объекта Stack
Peek()	Возвращает объект, находящийся на вершине объекта Stack, не удаляя его
Pop()	Возвращает объект, находящийся на вершине объекта Stack, и удаляет его из стека
Push()	Помещает объект на вершину объекта Stack
ToArray()	Копирует элементы объекта Stack в новый массив

Классы ArrayList, Queue и Stack имеют перегружаемые методы CopyTo() и ToArray(), позволяющие копировать элементы в массив. В случае класса Stack метод CopyTo() будет копировать элементы в существующий одномерный массив, замещая его содержимое начиная с указанного индекса. Метод ToArray() возвращает новый массив, содержащий элементы стека. Все это иллюстрируется в примере 9.16.

Пример 9.16. Работа со стеком

```
namespace Programming_CSharp
{
   using System;
   using System.Collections;

   // простой класс для хранения в массиве
   public class Tester
   {
      static void Main()
      {
         Stack intStack = new Stack();

         // заполнение стека
         for (int i = 0;i<8;i++)
         {
            intStack.Push(i*5);
         }
         // вывод стека на экран
         Console.Write( "Значения intStack:\t" );
         PrintValues( intStack );

         // удаление элемента из стека
         Console.WriteLine( "\n(Pop)\t{0}",
            intStack.Pop() );

         // вывод стека на экран
         Console.Write( "Значения intStack:\t" );
         PrintValues( intStack );

         // удаление еще одного элемента из стека
         Console.WriteLine( "\n(Pop)\t{0}",
            intStack.Pop() );

         // вывод стека на экран
         Console.Write( "Значения intStack:\t" );
         PrintValues( intStack );

         // прочитать элемент в вершине стека,
         // не удаляя его
         Console.WriteLine( "\n(Peek)   \t{0}",
            intStack.Peek() );

         // вывод стека на экран
         Console.Write( "Значения intStack:\t" );
         PrintValues( intStack );
```

```
            // объявить объект массива на 12 целых чисел
            Array targetArray=Array.CreateInstance(
               typeof(int), 12  );

            targetArray.SetValue( 100, 0 );
            targetArray.SetValue( 200, 1 );
            targetArray.SetValue( 300, 2 );
            targetArray.SetValue( 400, 3 );
            targetArray.SetValue( 500, 4 );
            targetArray.SetValue( 600, 5 );
            targetArray.SetValue( 700, 6 );
            targetArray.SetValue( 800, 7 );
            targetArray.SetValue( 900, 8 );

            // вывести значения элементов массива
            Console.WriteLine( "\nИсходный массив:  ");
            PrintValues( targetArray );

           // скопировать весь исходный стек  в новый массив начиная с индекса 6
            intStack.CopyTo( targetArray, 6 );

            // вывести значения нового массива
            Console.WriteLine( "\nИсходный массив после копирования:  ");
            PrintValues( targetArray );

            // скопировать весь исходный стек в новый стандартный массив
            Object[] myArray = intStack.ToArray();

            // вывести значения нового стандартного массива
            Console.WriteLine( "\nНовый массив:" );
            PrintValues( myArray );
         }

         public static void PrintValues(
            IEnumerable myCollection )
         {
            System.Collections.IEnumerator enumerator =
               myCollection.GetEnumerator();
            while ( enumerator.MoveNext() )
               Console.Write( "{0}  ",enumerator.Current );
            Console.WriteLine();
         }
      }
   }
```

Вывод:

```
Значения intStack:        35  30  25  20  15  10  5  0

(Pop)   35
Значения intStack:        30  25  20  15  10  5  0

(Pop)   30
Значения intStack:        25  20  15  10  5  0

(Peek)          25
Значения intStack:        25  20  15  10  5  0
```

```
Исходный массив:
100  200  300  400  500  600  700  800  900  0  0  0

Исходный массив после копирования:
100  200  300  400  500  600  25  20  15  10  5  0

Новый массив:
25  20  15  10  5  0

25
```

Выводимый программой текст иллюстрирует тот факт, что элементы, добавляемые в стек, извлекаются из него в обратном порядке. Фактически элементы стека хранятся в памяти в обратном порядке, отражая принцип LIFO.

В примере 9.16 используется класс `Array`, являющийся базовым для всех массивов. В частности, массив из 12 целых чисел создается с помощью статического метода `CreateInstance()`. Этот метод имеет два аргумента: тип элементов (в данном случае `int`) и число, обозначающее размер будущего массива.

Массив заполняется с помощью метода `SetValue()`, тоже имеющего два аргумента: добавляемый объект и смещение, по которому он должен находиться.

Эффект, производимый методом `CopyTo()`, можно понять, изучая обрабатываемый им массив до и после вызова метода. Элементы массива замещаются элементами стека начиная с указанного индекса (`6`).

Обратите также внимание на то, что метод `ToArray()` должен возвращать массив объектов. Поэтому `myArray` и объявляется соответственно:

```
Object[] myArray = intStack.ToArray();
```

Словари

Словарь (*dictionary*) – это класс коллекции, связывающий *ключ* со *значением*. По такому же принципу построены толковые словари, например словарь Вебстера связывает слово (ключ) с его толкованием (значение).

Чтобы понять ценность словарей как классов коллекций, представим, что требуется вести список столиц американских штатов. Один подход состоит в размещении их в массиве:

```
string[] stateCapitals = new string[50];
```

Массив `stateCapitals` будет содержать 50 названий столиц. Обращаться к каждой столице приходится по индексу. Например, чтобы прочитать название столицы штата Арканзас, необходимо знать, что Арканзас стоит на четвертом месте в алфавитном списке штатов:

```
string capitalOfArkansas = stateCapitals[3];
```

Это очень неудобный подход. Не так-то просто выяснить, что, например, Массачусетс – двадцать первый штат в алфавитном списке.

Гораздо удобнее хранить название столицы вместе с названием штата. *Словарь* как раз и позволяет хранить значение (в данном случае название столицы) вместе с ключом (названием штата).

Словарь платформы .NET Framework позволяет связывать любой ключ (строку, целочисленное значение, объект и т. д.) со значением любого типа (со строкой, целочисленным значением, объектом и т. д.). Конечно, чаще всего ключ довольно короткий, а значение – весьма сложное.

Самыми важными качествами хорошего словаря являются простота добавления новых записей и быстрота получения значений. Некоторые словари организованы так, что записи добавляются в них очень быстро; другие настроены на быстрое извлечение информации. Одним из примеров словаря является хеш-таблица.

Хеш-таблицы

Хеш-таблица (*hashtable*) – это словарь, оптимизированный для максимально быстрого получения информации. Основные методы и свойства класса `Hashtable` сведены в табл. 9.6.

Таблица 9.6. Методы и свойства класса Hashtable

Метод или свойство	Предназначение
`Synchronized()`	Открытый статический метод, возвращающий оболочку типа `Hashtable`, обеспечивающую безопасную работу с несколькими потоками (см. главу 20)
`Count`	Открытое свойство, позволяющее узнать текущее число элементов `Hashtable`
`IsReadOnly`	Открытое свойство, позволяющее выяснить, доступен ли объект `Hashtable` только для чтения
`IsSynchronized`	Открытое свойство, позволяющее выяснить, обеспечивает ли объект `Hash table` безопасность при работе с несколькими потоками
`Item()`	Индексатор для класса `Hashtable`
`Keys`	Открытое свойство, возвращающее `ICollection` с ключами класса `Hashtable`. (См. также `Values` далее в этой таблице)
`SyncRoot`	Открытое свойство, возвращающее объект, с помощью которого можно синхронизировать доступ к объекту `Hashtable`
`Values`	Открытое свойство, возвращающее `ICollection` со значениями класса `Hashtable`. (См. также `Keys` ранее в этой таблице)
`Add()`	Добавляет запись с указанной парой ключ/значение

Метод или свойство	Предназначение
Clear()	Удаляет все элементы из объекта Hashtable
Clone()	Создает копию объекта
Contains()ContainsKey()	Выясняет, содержит ли объект Hashtable указанный ключ
ContainsValue()	Выясняет, имеется ли в объекте Hashtable указанное значение
CopyTo()	Копирует объект Hashtable в существующий одномерный массив
GetEnumerator()	Возвращает перечислитель для объекта Hashtable
GetObjectData()	Реализует интерфейс ISerializable и возвращает данные, необходимые для сохранения объекта Hashtable в потоке
OnDeserialization()	Реализует интерфейс ISerializable и генерирует соответствующее событие при завершении восстановления состояния объекта из потока
Remove()	Удаляет запись с указанным ключом

В объекте Hashtable каждое значение хранится в блоках. Блоки пронумерованы, чем напоминают элементы массива.

Поскольку ключ может и не быть целым числом, нужен способ преобразования ключа (например строки «Массачусетс») в номер блока. Каждый ключ обязан предоставить метод GetHashCode(), выполняющий это действие.

Читатель, конечно, помнит, что в языке C# все ведет свое происхождение от класса Object. Этот класс имеет виртуальный метод GetHashCode(), который производные типы могут наследовать в неизменном виде, но могут и перегрузить.

Стандартная реализация метода GetHashCode() для строки сводится к тому, что Unicode-коды всех символов строки складываются, а затем с помощью деления по модулю получается значение от 0 до N, где N – количество блоков в хеш-таблице. Писать такой метод для типа string не надо, так как среда CLR предоставляет его по умолчанию.

Когда в объект Hashtable вставляются значения (названия столиц штатов), он вызывает метод GetHashCode() для каждого указанного ключа. Метод возвращает целочисленное значение, идентифицирующее блок, в который помещено значение.

Конечно, не исключена ситуация, когда для нескольких ключей будет возвращен один номер блока. Это называется *конфликтом* (*collision*). Существует ряд способов разрешения конфликтов. Самым распространенным подходом, к тому же принятым в CLR, является ведение упорядоченного списка значений в каждом блоке.

Чтобы прочитать значение из хеш-таблицы, программист указывает ключ. Объект Hashtable снова вызывает метод GetHashCode() для ключа и по возвращенному значению находит блок. Если в нем только одно значение, оно возвращается. Если несколько – выполняется двоичный поиск в содержимом блока. Поскольку в блоке обычно немного значений, поиск выполняется очень быстро.

Ключ хеш-таблицы может иметь базовый тип, а может быть экземпляром типа, определенного пользователем (объектом). Объекты, применяемые в качестве ключей хеш-таблицы, должны реализовать метод GetHashCode(), а также метод Equals(). В большинстве случаев достаточно просто воспользоваться реализацией, наследуемой от класса Object.

Интерфейс IDictionary

Хеш-таблицы являются словарями, поскольку они реализуют интерфейс IDictionary. Этот интерфейс предоставляет открытое свойство Item, которое возвращает значение по заданному ключу. В языке C# свойство Item объявляется так:

```
object this[object ключ]
{get; set;}
```

Свойство Item реализовано с помощью операции индексации ([ ]). Таким образом, к элементам любого объекта Dictionary можно обращаться, пользуясь привычным синтаксисом обращения к элементам массива.

В примере 9.17 демонстрируется добавление элементов в хеш-таблицу и последующее чтение их с помощью свойства Item.

Пример 9.17. Обращение к свойству Item с использованием смещения

```
namespace Programming_CSharp
{
   using System;
   using System.Collections;
   // простой класс для хранения в массиве
   public class Tester
   {
      static void Main()
      {
         // создать и инициализировать новую хеш-таблицу
         Hashtable hashTable = new Hashtable();
         hashTable.Add("000440312", "Jesse Liberty");
         hashTable.Add("000123933", "Stacey Liberty");
         hashTable.Add("000145938", "John Galt");
         hashTable.Add("000773394", "Ayn Rand");

         // обратиться к конкретному элементу
         Console.WriteLine("hashTable["000145938"]: {0}",
            hashTable["000145938"]);
      }
```

Коэффициент загрузки

Хеш-функции, минимизирующие число конфликтов, обычно добиваются этого ценой снижения эффективности использования памяти. Разработчикам хеш-функций приходится искать компромисс между минимизацией числа конфликтов, повышением эффективности использования памяти и скоростью работы алгоритма.

Для каждой хеш-таблицы среды CLR существует *коэффициент загрузки* (*load factor*), который определяет максимальное значение отношения количества записей к количеству блоков. Чем меньше этот коэффициент, тем выше производительность, но тем больше расходуется памяти. По умолчанию коэффициент загрузки равен 1.0, что, по утверждению фирмы Microsoft, обеспечивает оптимальный баланс между скоростью и размером. Однако программист, создающий экземпляр класса `Hashtable`, может задать любой коэффициент загрузки.

По мере добавления новых записей в объект `Hashtable` фактическая загрузка увеличивается, пока она не сравняется с коэффициентом, указанным во время создания объекта. Достигнув заданного коэффициента, объект `Hashtable` автоматически увеличивает количество блоков до наименьшего простого числа, большего, чем удвоенное текущее количество блоков.

Если какая-либо хеш-таблица имеет только один блок, поиск значения по ключу будет обыкновенным двоичным поиском, и хеш-таблица вообще окажется ненужной. Если же объект `Hashtable` содержит несколько блоков, процедура будет заключаться в вычислении хеш-функции ключа, поиске нужного блока и поиске внутри блока. Если у блока только один ключ, поиск происходит очень быстро.

Если ключей очень много, потребуется немало времени на поиск нужного блока или на поиск внутри него. За достижение компромисса как раз и отвечает поле `LoadFactor`.

```
    }
}
```

Вывод:

```
hashTable["000145938"]: John Galt
```

Пример 9.17 начинается с создания экземпляра `Hashtable`. Для этого вызывается простейший конструктор, который использует установленные по умолчанию значения для исходного размера, коэффициента загрузки (о коэффициенте загрузки см. врезку «Коэффициент загрузки»), метода получения хеш-кода и процедуры сравнения.

Далее в таблицу записываются четыре пары ключ/значение. В данном примере номер социальной страховки связан с именем и фамилией (обратите внимание, что здесь намеренно взяты выдуманные номера страховок).

После добавления элементов происходит чтение третьего из них.

Коллекции Keys и Values

Класс коллекции Dictionary предоставляет два дополнительных свойства, Keys и Values. Первое из них возвращает объект ICollection, содержащий все ключи хеш-таблицы, а второе – объект ICollection со всеми значениями. Пример 9.18 иллюстрирует сказанное.

Пример 9.18. Коллекции Keys и Values

```
namespace Programming_CSharp
{
   using System;
   using System.Collections;

   // простой класс для хранения в массиве
   public class Tester
   {
      static void Main()
      {
         // создать и инициализировать новую хеш-таблицу
         Hashtable hashTable = new Hashtable();
         hashTable.Add("000440312", "George Washington");
         hashTable.Add("000123933", "Abraham Lincoln");
         hashTable.Add("000145938", "John Galt");
         hashTable.Add("000773394", "Ayn Rand");

         // прочитать все ключи хеш-таблицы
         ICollection keys = hashTable.Keys;

         // прочитать все значения
         ICollection values = hashTable.Values;

         // перебрать в цикле коллекцию ключей
         foreach(string key in keys)
         {
            Console.WriteLine("{0} ", key);
         }

         // перебрать в цикле коллекцию значений
         foreach (string val in values)
         {
            Console.WriteLine("{0} ", val);
         }
      }
   }
}
```

Вывод:

```
000440312
000123933
000773394
000145938
George Washington
Abraham Lincoln
Ayn Rand
John Galt
```

Хотя порядок следования элементов в коллекции `Keys` не гарантируется, можно *гарантировать*, что он соответствует порядку следования элементов в коллекции `Values`.

Интерфейс IDictionaryEnumerator

Объект `IDictionary` поддерживает конструкцию `foreach` благодаря реализации метода `GetEnumerator()`, возвращающего объект типа `IDictionaryEnumerator`.

Объекты `IDictionaryEnumerator` используются для перебора объектов `IDictionary`. Они обладают свойствами, предоставляющими доступ к ключам и значениям каждого из элементов словаря. Пример 9.19 демонстрирует это.

Пример 9.19. Использование интерфейса IDictionaryEnumerator

```
namespace Programming_CSharp
{
   using System;
   using System.Collections;

   // простой класс для хранения в массиве
   public class Tester
   {

      static void Main()
      {
         // создать и инициализировать новую хеш-таблицу
         Hashtable hashTable = new Hashtable();
         hashTable.Add("000440312", "George Washington");
         hashTable.Add("000123933", "Abraham Lincoln");
         hashTable.Add("000145938", "John Galt");
         hashTable.Add("000773394", "Ayn Rand");

         // вывести свойства и значения хеш-таблицы
         Console.WriteLine( "hashTable" );
         Console.WriteLine( "  Счетчик:    {0}", hashTable.Count );
         Console.WriteLine( "  Ключи и значения:" );
         PrintKeysAndValues( hashTable );
      }

      public static void PrintKeysAndValues( Hashtable table )
```

```
        {
            IDictionaryEnumerator enumerator = table.GetEnumerator();
            while ( enumerator.MoveNext() )
                Console.WriteLine( "\t{0}:\t{1}",
                    enumerator.Key, enumerator.Value );
            Console.WriteLine();

        }
    }
}
```

Вывод:

```
hashTable
  Счетчик:    4
  Ключи и значения:
        000440312:      George Washington
        000123933:      Abraham Lincoln
        000773394:      Ayn Rand
        000145938:      John Galt
```

10

Строки и регулярные выражения

Раньше люди применяли компьютеры только для обработки числовых значений. На компьютерах рассчитывались траектории полета ракет, а программирование преподавалось лишь на математических факультетах крупнейших университетов.

Сегодня большинство программ имеет дело не с числами, а с символьными строками. Типичными примерами служат редактирование текстов, обработка документов и создание веб-страниц.

Язык C# предоставляет встроенную поддержку полноценного типа `string`. Более того, в C# строки трактуются как объекты, инкапсулирующие методы обработки, сортировки и поиска, обычно применяемые к символьным строкам.

Более сложные манипуляции со строками и поиск соответствия по шаблону выполняются с использованием *регулярных выражений* (*regular expressions*). Язык C# сочетает в себе всю мощь и сложность синтаксиса регулярных выражений, ранее характерную только для специализированных языков, таких как awk и Perl, с полностью объектно-ориентированным подходом к разработке.

В этой главе вы узнаете, как работать с типом `string` языка C# и классом `System.String` платформы .NET Framework, для которого `string` является псевдонимом. Узнаете, как извлекать фрагменты строк, обрабатывать строки и объединять их, а также как создавать новые строки с помощью класса `StringBuilder`. Кроме того, будет продемонстрировано, как пользоваться классом `RegEx` для обработки строк при помощи сложных регулярных выражений.

Строки

В языке C# строки являются полноценными типами, гибкими, мощными и простыми в обращении. Каждый объект string – это *неизменяемая* (*immutable*) последовательность символов Unicode. Иными словами, методы, предназначенные для изменения строк, возвращают измененные копии, исходные же строки остаются неизменными.

Объявляя строку в языке C# с помощью ключевого слова string, программист фактически объявляет объект типа System.String, одного из базовых типов, предоставляемых библиотекой классов платформы .NET Framework. В языке C# тип string *является* типом System.String, и в данной главе различие между этими именами не проводится.

Класс System.String объявляется следующим образом:

```
public sealed class String :
    IComparable, ICloneable, IConvertible, IEnumerable
```

Это объявление свидетельствует о том, что класс является изолированным, то есть от него нельзя произвести другой класс. Класс System.String реализует интерфейсы IComparable, ICloneable, IConvertible и IEnumerable, которые определяют функциональные возможности, разделяемые классом System.String с другими классами платформы .NET Framework.

Как было показано в главе 9, интерфейс IComparable реализуется типами, значения которых могут быть упорядочены. Так, строки могут быть упорядочены по алфавиту, и любую строку можно сравнить с любой другой для выяснения, которая из них стоит раньше в упорядоченном списке. Классы, реализующие интерфейс IComparable, реализуют метод CompareTo(). Интерфейс IEnumerable, также рассмотренный в главе 9, позволяет использовать цикл foreach для перебора строки как коллекции символов.

Объекты ICloneable способны создавать объекты с тем же значением, что и исходный объект. В данном случае клонирование строки приводит к созданию новой строки с теми же символами, что и у исходной. Классы, реализующие интерфейс ICloneable, реализуют метод Clone().

Если класс реализует интерфейс IConvertible, значит, он реализует методы, способствующие преобразованию в другие встроенные типы, например ToInt32(), ToDouble(), ToDecimal() и т. д.

Создание строк

Наиболее распространенным способом создания строки является присваивание *строкового литерала* (то есть последовательности символов, заключенной в кавычки) переменной типа string, определяемой пользователем:

```
string newString = "This is a string literal"
```

Строки, заключенные в кавычки, могут включать в себя *управляющие последовательности*, такие как «\t» или «\n», которые начинаются с символа обратной наклонной черты (обратного слэша) (\) и обозначают табуляцию или переход на новую строку. Поскольку обратная наклонная черта сама встречается в синтаксисе командных строк (например в адресах URL или в путях к каталогу), внутри строки ей должна предшествовать еще одна.

Строки можно создавать с помощью *дословных* (*verbatim*) строковых литералов, которые начинаются с символа @. Такой литерал сообщает конструктору String(), что строку следует воспринимать буквально, даже если она простирается на несколько строк программы или содержит управляющие последовательности. В дословном строковом литерале символы обратной наклонной черты и следующие за ними считаются обычными символами строки. Таким образом, следующие два определения эквивалентны:

```
string literalOne = "\\\\MySystem\\MyDirectory\\ProgrammingC#.cs;"
string verbatimLiteralOne = @"\\MySystem\MyDirectory\ProgrammingC#.cs";
```

В первой строчке кода использован обычный строковый литерал, поэтому каждый символ обратной наклонной черты имеет перед собой еще один. Во второй строке необходимость в дополнительных символах отпадает благодаря применению дословного литерала. Следующий код демонстрирует многострочные (занимающие более одной строчки текста) строки, одна из которых – дословная:

```
string literalTwo = "Line One\nLine Two";
string verbatimLiteralTwo = @"Line One
Line Two";
```

И опять эти определения эквивалентны. Какому отдать предпочтение – дело личного вкуса и удобства.

Метод ToString()

Еще одним общепринятым способом создания строк является вызов метода ToString() для объекта и присваивание результата строковой переменной. Все встроенные типы переопределяют этот метод, чтобы упростить преобразование значения (в большинстве случаев числового) в его строковое представление. В следующем участке кода метод ToString() вызывается для переменной типа int с целью записи ее в строку:

```
int myInteger = 5;
string integerString = myInteger.ToString()
```

Вызов myInteger.ToString() возвращает объект String, который присваивается строке integerString.

Класс `String` платформы .NET предоставляет программисту огромное количество перегружаемых конструкторов, поддерживающих различные технологии присваивания строковых значений переменным типа `string`. Некоторые из этих конструкторов позволяют программисту создавать строки, передавая в качестве параметра массив символов или указатель на символ. Передача массива конструктору объекта `String` приводит к созданию нового экземпляра строки, удовлетворяющего описанию CLR. Передача указателя на символ заканчивается созданием «небезопасного» экземпляра, не удовлетворяющего требованиям CLR.

Действия со строками

Класс `string` обладает богатым набором методов для сравнения строк, поиска в строке и других действий со строками. Эти методы перечислены в табл. 10.1.

Таблица 10.1. Методы и поля класса string

Метод или поле	Пояснение
`Empty`	Открытое статическое поле, представляющее пустую строку
`Compare()`	Перегруженный открытый статический метод, сравнивающий две строки
`CompareOrdinal()`	Перегруженный открытый статический метод, сравнивающий две строки без учета языка и культуры
`Concat()`	Перегруженный открытый статический метод, создающий новую строку из нескольких строк
`Copy()`	Открытый статический метод, создающий новую строку путем копирования существующей строки
`Equals()`	Перегруженный открытый статический метод и метод экземпляра, производящий сравнение двух строк на равенство
`Format()`	Перегруженный открытый статический метод, форматирующий строку в соответствии со спецификацией формата
`Intern()`	Открытый статический метод, возвращающий ссылку на указанный объект строки
`IsInterned()`	Открытый статический метод, возвращающий ссылку для строки
`Join()`	Перегруженный открытый статический метод, преобразующий элементы массива в одну строку
`Chars`	Индексатор строки
`Length`	Количество символов в данном экземпляре
`Clone()`	Возвращает строку
`CompareTo()`	Сравнивает данную строку с другой

Метод или поле	Пояснение
CopyTo()	Копирует указанное количество символов в массив символов Unicode
EndsWith()	Показывает, соответствует ли указанная строка окончанию данной строки
Equals()	Выясняет, одинаковые ли значения у двух строк
Insert()	Возвращает новую строку, в которую вставлена указанная строка
LastIndexOf()	Возвращает индекс последнего вхождения указанного символа или подстроки в данную строку
PadLeft()	Выравнивает символы строки по правой границе, заполняя свободное место слева пробелами или указанными символами
PadRight()	Выравнивает символы строки по левой границе, заполняя свободное место справа пробелами или указанными символами
Remove()	Удаляет указанное количество символов
Split()	Возвращает массив из фрагментов строки, разделенных в исходной строке указанными символами
StartsWith()	Сообщает, начинается ли строка с указанных символов
Substring()	Извлекает фрагмент строки
ToCharArray()	Копирует символы из строки в массив символов
ToLower()	Возвращает копию строки, преобразованную в нижний регистр
ToUpper()	Возвращает копию строки, преобразованную в верхний регистр
Trim()	Удаляет все вхождения указанных символов в начале и конце строки
TrimEnd()	Ведет себя как Trim(), но только в отношении конца строки
TrimStart()	Ведет себя как Trim(), но только в отношении начала строки

В примере 10.1 иллюстрируется применение некоторых из этих методов, в том числе Compare(), Concat() (и перегруженной операции «+»), Copy() (и операции «=»), Insert(), EndsWith() и IndexOf().

Пример 10.1. Работа со строками

```
namespace Programming_CSharp
{
   using System;

   public class StringTester
   {
      static void Main()
      {
         // создать несколько строк для последующей работы
```

```
string s1 = "abcd";
string s2 = "ABCD";
string s3 = @"Liberty Associates, Inc.
       provides custom .NET development,
       on-site Training and Consulting";

int result;  // для результатов сравнения

// сравнить две строки с учетом регистра символов
result = string.Compare(s1, s2);
Console.WriteLine(
   "сравнение s1: {0}, s2: {1}, результат: {2}\n",
   s1, s2, result);

// перегруженное сравнение, принимает логическое
// значение "игнорировать регистр" (= true)
result = string.Compare(s1,s2, true);
Console.WriteLine("сравнение без учета регистра\n");
Console.WriteLine("s4: {0}, s2: {1}, результат: {2}\n",
   s1, s2, result);

// метод для объединения строк
string s6 = string.Concat(s1,s2);
Console.WriteLine(
   "s6 объединение строк s1 и s2: {0}", s6);

// использовать перегруженную операцию
string s7 = s1 + s2;
Console.WriteLine(
   "s7 объединение строк s1 + s2: {0}", s7);

// метод для копирования строк
string s8 = string.Copy(s7);
Console.WriteLine(
   "s8 копируется из s7: {0}", s8);

// использовать перегруженную операцию
string s9 = s8;
Console.WriteLine("s9 = s8: {0}", s9);

// три способа выполнить сравнение
Console.WriteLine(
   "\nПроверка s9.Equals(s8): {0}",
   s9.Equals(s8));
Console.WriteLine(
   "Проверка Equals(s9,s8): {0}",
   string.Equals(s9,s8));
Console.WriteLine(
   "Проверка s9==s8: {0}", s9 == s8);

// два полезных свойства: индекс и длина (Length)
Console.WriteLine(
   "\nДлина строки s9 {0} символов. ",
   s9.Length);
Console.WriteLine(
```

```
            "Пятый символ {1}\n",
            s9.Length, s9[4]);

         // проверить, заканчивается ли строка указанными символами
         Console.WriteLine("s3:{0}\nЗаканчивается строкой Training?: {1}\n",
            s3,
            s3.EndsWith("Training") );
         Console.WriteLine(
            "Заканчивается строкой Consulting?: {0}",
            s3.EndsWith("Consulting"));

         // возвратить индекс фрагмента строки
         Console.WriteLine(
            "\nпервое появление строки Training ");
         Console.WriteLine ("в s3 - {0}\n",
            s3.IndexOf("Training"));

         // вставить слово "excellent" перед словом "training"
         string s10 = s3.Insert(101,"excellent ");
         Console.WriteLine("s10: {0}\n",s10);

         // можно объединить две операции в одну
         string s11 = s3.Insert(s3.IndexOf("Training"),
            "excellent ");
         Console.WriteLine("s11: {0}\n",s11);
      }
   }
}
```

Вывод:

```
сравнение s1: abcd, s2: ABCD, результат: -1

сравнение без учета регистра

s4: abcd, s2: ABCD, результат: 0

s6 объединение строк s1 и s2: abcdABCD
s7 объединение строк s1 + s2: abcdABCD
s8 копируется из s7: abcdABCD
s9 = s8: abcdABCD

Проверка s9.Equals(s8)?: True
Проверка Equals(s9,s8)?: True
Проверка s9==s8?: True

Длина строки s9 8 символов.
Пятый символ A

s3:Liberty Associates, Inc.
                provides custom .NET development,
                on-site Training and Consulting
Заканчивается строкой Training?: False
Заканчивается строкой Consulting?: True

Первое появление строки Training
в s3 - 101
```

```
s10: Liberty Associates, Inc.
                provides custom .NET development,
                on-site excellent Training and Consulting

s11: Liberty Associates, Inc.
                provides custom .NET development,
                on-site excellent Training and Consulting
```

Пример 10.1 начинается с объявления трех строк:

```
string s1 = "abcd";
string s2 = "ABCD";
string s3 = @"Liberty Associates, Inc.
      provides custom .NET development,
      on-site Training and Consulting";
```

Первые две определяются обычными строковыми литералами, третья – дословным строковым литералом. Сначала выполняется сравнение `s1` и `s2`. Метод `Compare()` является открытым статическим методом типа `string`, причем он перегружается. Первая перегруженная версия принимает две строки и сравнивает их:

```
// сравнить две строки с учетом регистра
result = string.Compare(s1, s2);
Console.WriteLine("сравнение s1: {0}, s2: {1}, результат: {2}\n",
    s1, s2, result);
```

Этот метод учитывает регистр и возвращает значение, зависящее от результатов сравнения:

- Отрицательное целое, если первая строка меньше второй
- Ноль, если строки равны
- Положительное целое, если первая строка больше второй

В данном случае вывод отражает тот факт, что `s1` «меньше», чем `s2`. В кодировке Unicode (как и в ASCII) буквы нижнего регистра имеют меньшие коды, чем буквы верхнего:

```
сравнение s1: abcd, s2: ABCD, результат: -1
```

При втором сравнении задействована перегруженная версия метода `Compare()`, которая принимает еще и третий параметр – логическое значение, определяющее, следует ли при сравнении игнорировать регистр. Если этот параметр равен `true`, сравнение делается без учета регистра:

```
result = string.Compare(s1,s2, true);
Console.WriteLine("сравнение без учета регистра\n");
Console.WriteLine("s4: {0}, s2: {1}, результат: {2}\n", s1, s2, result);
```

Результат выведен с помощью двух вызовов метода `WriteLine()`, чтобы строчки уместились на странице книги.

На этот раз регистр игнорируется, и результатом сравнения будет 0. Две строки идентичны, если не принимать во внимание разницу в регистре символов:

```
сравнение без учета регистра

s4: abcd, s2: ABCD, результат: 0
```

Далее в примере 10.2 выполняется объединение строк. Для этого существует два способа. Можно вызвать Concat(), открытый статический метод типа string:

```
string s6 = string.Concat(s1,s2);
```

либо воспользоваться перегруженной операцией +:

```
string s7 = s1 + s2;
```

В обоих случаях выведенные строки демонстрируют успешное завершение операции объединения строк:

```
s6 объединение строк s1 и s2: abcdABCD
s7 объединение строк s1 + s2: abcdABCD
```

Аналогично создание копии строки выполняется двумя способами. Во-первых, вызовом метода Copy():

```
string s8 = string.Copy(s7);
```

а во-вторых, что гораздо удобнее, можно воспользоваться перегруженной операцией присваивания (=), которая неявно создает копию строки:

```
string s9 = s8;
```

И опять выводимые строки подтверждают, что оба способа корректны:

```
s8 копируется из s7: abcdABCD
s9 = s8: abcdABCD
```

Класс String платформы .NET предоставляет три способа проверки равенства строк. Первый – вызов перегруженного метода Equals() и непосредственный запрос строке s9 о том, содержит ли s8 аналогичное значение:

```
Console.WriteLine("\nПроверка s9.Equals(s8): {0}", s9.Equals(s8));
```

Второй способ состоит в передаче сравниваемых строк статическому методу Equals() класса String:

```
Console.WriteLine("Проверка Equals(s9,s8): {0}", string.Equals(s9,s8));
```

Последний способ заключается в использовании перегруженной операции проверки на равенство (==) класса String:

```
Console.WriteLine("Проверка s9==s8: {0}", s9 == s8);
```

В каждом из трех случаев возвращается логическое значение, что видно из вывода:

```
Проверка s9.Equals(s8): True
Проверка Equals(s9,s8): True
Проверка s9==s8: True
```

Операция сравнения выглядит наиболее естественно при наличии двух строковых объектов. Однако в некоторых языках, например в VB.NET, отсутствует поддержка перегрузки операций, поэтому программист должен обязательно перегрузить метод `Equals()` для экземпляра.

В следующих строках примера 10.1 операция индексирования (`[]`) используется для поиска конкретного символа внутри строки, а свойство `Length` – для выяснения длины строки:

```
Console.WriteLine("\nДлина строки s9 {0} символов., s9.Length);
Console.WriteLine("Пятый символ {1}\n", s9.Length, s9[4]);
```

Вывод:

```
Длина строки s9 8 символов.
Пятый символ A
```

Метод `EndsWith()` спрашивает строку, заканчивается ли она указанной строкой. Так, у строки `s3` можно спросить, заканчивается ли она словом «Training» (что на самом деле не так), а затем – заканчивается ли словом «Consulting» (что действительно имеет место):

```
// проверить, заканчивается ли строка указанными символами
Console.WriteLine("s3:{0}\nЗаканчивается строкой Training?: {1}\n",
    s3, s3.EndsWith("Training") );
Console.WriteLine("Заканчивается строкой Consulting?: {0}",
    s3.EndsWith("Consulting"));
```

Из выведенного текста понятно, что первая проверка дала отрицательный результат, а вторая – положительный:

```
s3:Liberty Associates, Inc.
              provides custom .NET development,
              on-site Training and Consulting
Заканчивается строкой Training?: False
Заканчивается строкой Consulting?: True
```

Метод `IndexOf()` находит в строке указанный фрагмент, а метод `Insert()` заменяет этот фрагмент в копии исходной строки.

В следующем участке программы определяется положение первого вхождения слова «Training» в строке `s3`:

```
Console.WriteLine("\nПервое появление строки Training ");
Console.WriteLine ("в s3 - {0}\n", s3.IndexOf("Training"));
```

Выведенные строки свидетельствуют, что смещение равно `103`:

```
Первое появление строки Training
в s3 - 103
```

Теперь в строку можно вставлять слово «excellent» с пробелом после него. Вставка будет выполнена в копию исходной строки, возвращаемую методом Insert(). Строке s10 присваивается именно копия:

```
string s10 = s3.Insert(103,"excellent ");
Console.WriteLine("s10: {0}\n",s10);
```

В результате выполнения этого фрагмента программы выводится следующий текст:

```
s10: Liberty Associates, Inc.
provides custom .NET development,
on-site excellent Training and Consulting
```

Наконец, можно объединить все эти операции в одном эффективном операторе:

```
string s11 = s3.Insert(s3.IndexOf("Training"),"excellent ");
Console.WriteLine("s11: {0}\n",s11);
```

Выведенный текст практически идентичен:

```
s11: Liberty Associates, Inc.
provides custom .NET development,
on-site excellent Training and Consulting
```

Поиск фрагментов строк

Класс String предоставляет перегруженный метод Substring() для вырезания фрагментов из строк. Обе имеющиеся версии этого метода используют индекс, показывающий, откуда начинать вырезание подстроки, а одна из версий использует также индекс, определяющий конец вырезаемой подстроки. Применение метода Substring() проиллюстрировано в примере 10.2.

Пример 10.2. Применение метода Substring( )

```
namespace Programming_CSharp
{
   using System;
   using System.Text;

   public class StringTester
   {
      static void Main()
      {
         // создать несколько строк для последующей работы
         string s1 = "One Two Three Four";

         int ix;

          // получить индекс последнего пробела
         ix=s1.LastIndexOf(" ");
```

```
            // получить последнее слово
            string s2 = s1.Substring(ix+1);

            // присвоить строке s1 фрагмент строки, начиная с 0
            // и заканчивая индексом ix (началом последнего слова)
            // таким образом, в s1 будет "One Two Three"
            s1 = s1.Substring(0,ix);

            // найти последний пробел в s1 (после "Two")
            ix = s1.LastIndexOf(" ");

            // присвоить строке s3 фрагмент строки, начиная с позиции ix
            // и заканчивая позицией последнего пробела плюс 1,
            // то есть s3 = "Three"
            string s3 = s1.Substring(ix+1);

            // присвоить строке s1 фрагмент строки, начиная с 0
            // и заканчивая индексом ix
            // таким образом, в s1 будет "One Two"
            s1 = s1.Substring(0,ix);

            // присвоить переменной ix позицию пробела между
            // "One" и "Two"
            ix = s1.LastIndexOf(" ");

            // присвоить строке s4 фрагмент строки, начиная с позиции,
            // следующей за ix, то есть "Two"
            string s4 = s1.Substring(ix+1);

            // присвоить строке s1 фрагмент строки, начиная с 0
            // и заканчивая индексом ix
            // таким образом, в s1 будет "One"
            s1 = s1.Substring(0,ix);

            // присвоить переменной ix позицию последнего пробела,
            // но его нет, поэтому ix = -1
            ix = s1.LastIndexOf(" ");

            // присвоить строке s5 фрагмент строки, начиная с позиции,
            // следующей за последним пробелом;
            // поскольку он отсутствует, строке s5
            // присваивается фрагмент строки, начинающийся с 0
            string s5 = s1.Substring(ix+1);

            Console.WriteLine ("s2: {0}\ns3: {1}",s2,s3);
            Console.WriteLine ("s4: {0}\ns5: {1}\n",s4,s5);
            Console.WriteLine ("s1: {0}\n",s1);

        }
    }
}
```

Вывод:

```
s2: Four
s3: Three
s4: Two
```

```
s5: One

s1: One
```

В примере 10.2 представлено далеко не самое элегантное решение задачи извлечения слов из строки. Тем не менее, он годится в качестве первого приближения и демонстрирует полезные технические приемы. Пример начинается с создания строки s1:

```
string s1 = "One Two Three Four";
```

Переменной ix присваивается индекс *последнего* пробела в строке:

```
ix=s1.LastIndexOf(" ");
```

Затем фрагмент строки, начинающийся с позиции, следующей за найденным пробелом, присваивается новой строке s2:

```
string s2 = s1.Substring(ix+1);
```

Здесь фрагмент строки извлекается начиная с ix+1 и до конца строки, то есть строке s2 присваивается значение «Four».

Следующий шаг заключается в удалении слова «Four» из строки s1. Это можно сделать, присвоив строке s1 ее же фрагмент от начала строки до символа в позиции ix:

```
s1 = s1.Substring(0,ix);
```

Переменной ix присваивается позиция последнего (из оставшихся) пробела, стоящего перед словом «Three», извлекаемым затем в строку s3. Аналогично заполняются строки s4 и s5. В конце работы программы на экран выводятся результаты:

```
s2: Four
s3: Three
s4: Two
s5: One

s1: One
```

Не элегантно, но работает и, к тому же, иллюстрирует действие метода Substring(). Такой подход несколько напоминает арифметику указателей C++, но без указателей и без потери безопасности кода.

Разбиение строк

Более эффективное решение задачи, поставленной в примере 10.2, заключается в применении метода Split() класса String. Назначение этого метода в том и состоит, чтобы разбивать строку на подстроки. Вызывая метод Split(), следует передать ему массив разделителей (символов, стоящих между словами), и метод возвратит массив фрагментов строк. Пример 10.3 иллюстрирует работу этого метода.

Пример 10.3. Применение метода Split( )

```
namespace Programming_CSharp
{
   using System;
   using System.Text;

   public class StringTester
   {
      static void Main()
      {
         // создать несколько строк для последующей работы
         string s1 = "One,Two,Three Liberty Associates, Inc.";

         // константы для пробела и запятой
         const char Space = ' ';
         const char Comma = ',';

         // массив разделителей, разбивающих предложение
         char[] delimiters = new char[]
            {
               Space,
               Comma
            };

         string output = "";
         int ctr = 1;

         // разбить строку и в цикле перебрать элементы массива
         // фрагментов строк
         foreach (string subString in s1.Split(delimiters))
         {
            output += ctr++;
            output += ": ";
            output += subString;
            output += "\n";
         }
         Console.WriteLine(output);
      }
   }
}
```

Вывод:

```
1: One
2: Two
3: Three
4: Liberty
5: Associates
6:
7: Inc.
```

Пример начинается с создания строки, подлежащей разбиению:

```
string s1 = "One,Two,Three Liberty Associates, Inc.";
```

Символами-разделителями будут пробел и запятая. Для строки s1 вызывается метод Split(), а результаты передаются оператору foreach:

```
foreach (string subString in s1.Split(delimiters))
```

Строка output инициализируется пустой строкой и затем формируется за четыре шага. Вначале берется значение ctr. Затем добавляется двоеточие, далее – фрагмент строки, возвращенный методом Split(), и, наконец, символ перевода строки. Эти четыре шага повторяются для каждого фрагмента строки, возвращенного методом Split(). Надо признать, такое повторяющееся копирование чрезвычайно неэффективно.

Проблема в том, что строковый тип не предназначен для подобной операции. На самом деле требуется, чтобы при каждом проходе цикла создавалась новая строка путем присоединения отформатированной строки. Для этой цели и предназначен класс StringBuilder.

Манипуляции с динамическими строками

Класс System.Text.StringBuilder используется для создания строк и их модификации. Семантически это инкапсуляция конструктора для класса String. Важнейшие члены класса StringBuilder сведены в табл. 10.2.

Таблица 10.2. Методы класса StringBuilder

Метод	Пояснение
Capacity()	Читает или изменяет число символов, которое может содержать объект StringBuilder
Chars()	Индексатор
Length()	Читает или изменяет длину объекта StringBuilder
MaxCapacity()	Читает максимальное количество символов, которое может вместить объект StringBuilder
Append()	Перегруженный открытый метод, присоединяющий типизированный объект к концу текущего объекта StringBuilder
AppendFormat()	Перегруженный открытый метод, заменяющий спецификаторы формата на отформатированное значение объекта
EnsureCapacity()	Гарантирует, что текущий объект StringBuilder имеет размер (число символов, которое может быть в него помещено), как минимум равный указанному значению
Insert()	Перегруженный открытый метод, вставляющий объект в указанную позицию
Remove()	Удаляет указанные символы
Replace()	Перегруженный открытый метод, заменяющий все вхождения указанных символов на новые символы

В отличие от класса String, класс StringBuider изменяемый. При внесении изменений в экземпляр этого класса меняется сама строка, а не ее

копия. В примере 10.4 объект `String` из примера 10.3 заменен на объект `StringBuider`.

Пример 10.4. Использование класса StringBuider

```
namespace Programming_CSharp
{
   using System;
   using System.Text;

   public class StringTester
   {
      static void Main()
      {
         // создать несколько строк для последующей работы
         string s1 = "One,Two,Three Liberty Associates, Inc.";

         // константы для пробела и запятой
         const char Space = ' ';
         const char Comma = ',';

         // массив разделителей, разбивающих предложение
         char[] delimiters = new char[]
         {
               Space,
               Comma
         };

         // построить выходную строку output с помощью
         // конструктора StringBuilder()
         StringBuilder output = new StringBuilder();
         int ctr = 1;

         // разбить строку и в цикле перебрать массив фрагментов строк
         foreach (string subString in s1.Split(delimiters))
         {
            // метод AppendFormat присоединяет отформатированную строку
            output.AppendFormat("{0}: {1}\n",ctr++,subString);
         }
         Console.WriteLine(output);
      }
   }
}
```

Переделана лишь последняя часть предыдущей программы. Теперь изменения в строку вносятся не с помощью объединения строк, а методом `AppendFormat()` класса `StringBuilder`. Этот метод добавляет новые отформатированные строки по мере их создания, что намного проще и эффективнее. Результаты же идентичны предыдущим:

```
1: One
2: Two
3: Three
4: Liberty
5: Associates
```

```
6:
7: Inc.
```

Регулярные выражения

Регулярные выражения – это мощный язык для описания текста и внесения в него изменений. Регулярное выражение *применяется* к строке, то есть к последовательности символов. Нередко эта строка представляет собой весь текстовый документ.

Результатом применения регулярного выражения к строке является возвращение либо фрагмента строки, либо новой строки, которая представляет собой измененную часть исходной строки. Помните, что строки являются неизменяемыми объектами, так что регулярное выражение не способно изменить исходную строку.

Применив правильно сформулированное регулярное выражение к следующей строке:

```
One,Two,Three Liberty Associates, Inc.
```

можно получить в результате любой ее фрагмент (например, `Liberty` или `One`) или измененную версию какого-нибудь фрагмента строки (например, `LIBeRtY` или `OnE`). Что именно делает регулярное выражение, определяется структурой самого регулярного выражения.

Регулярное выражение состоит из символов двух видов: *литералов* и *метасимволов*. Литерал – это символ, для которого требуется найти соответствие в исходной строке. Метасимвол – это специальный символ, который служит своего рода инструкцией синтаксическому анализатору регулярного выражения. Синтаксический анализатор – это механизм, ответственный за интерпретацию регулярного выражения. Например, если написать регулярное выражение:

```
^(From|To|Subject|Date):
```

то ему будет соответствовать любой фрагмент исходной строки, состоящий из последовательности символов «From» или «To», или «Sub-

Ограничения, накладываемые на разделители

Поскольку в качестве разделителей были указаны запятая и пробел, то пробел после запятой между словами «Associates» и «Inc.» воспринят как отдельное слово. Естественно, не это имелось в виду. Чтобы исправить ситуацию, нужно сообщить методу `Split()`, что разделителем является или запятая (между `One`, `Two` и `Three`), или пробел (между `Liberty` и `Associates`), или запятая, за которой следует пробел. Последнее условие отнюдь не тривиально, и для его формулировки потребуется использовать регулярное выражение.

ject», или «Date», при условии, что этот фрагмент строки начинает новую строку текста (^) и заканчивается двоеточием (:).

Символ «^» в данном случае сообщает анализатору регулярного выражения, что искомый фрагмент строки должен находиться в начале строки текста. Последовательности символов в словах «From» и «To» являются литералами, а метасимволы открывающей и закрывающей скобок ((,)) и вертикальная черта (|) служат для группирования литералов и указания, что каждая группа должна быть рассмотрена при поиске. (Обратите внимание, что символ «^» тоже является метасимволом, обозначающим начало строки.)

Итак, выражение:

```
^(From|To|Subject|Date):
```

можно озвучить следующим образом: «Найти любую строку, которая начинается с одной из четырех литеральных строк (From, To, Subject или Date), за которыми следует двоеточие».

Детальное рассмотрение регулярных выражений выходит за пределы этой книги, однако все регулярные выражения, использованные в примерах, разъясняются достаточно подробно. Читателю, интересующемуся этой темой, можно порекомендовать книгу Джеффри Фридла «Регулярные выражения. Библиотека программиста». – Пер. с англ. – СПб: Питер, 2001 г.

Работа с регулярными выражениями. Класс Regex

Платформа .NET Framework реализует объектно-ориентированный подход к поиску и замене шаблонов, определяемых регулярными выражениями.

В основе регулярных выражений лежат выражения *regexp* языка Perl5, включающие в себя «ленивые» спецификаторы количества (**??**, ***?**, **+?**, **{n,m}?**), положительный и отрицательный просмотр, а также условное вычисление выражения.

Пространство имен библиотеки базовых классов (Base Class Library) System.Text.RegularExpressions содержит все объекты платформы .NET Framework, имеющие отношение к регулярным выражениям. Важнейшим классом, поддерживающим регулярные выражения, является Regex, который представляет неизменяемые откомпилированные регулярные выражения. Хотя класс Regex допускает создание экземпляров, он предоставляет программисту ряд полезных статических методов. Применение класса Regex демонстрируется в примере 10.5.

Пример 10.5. Использование класса Regex при работе с регулярными выражениями

```
namespace Programming_CSharp
{
   using System;
   using System.Text;
   using System.Text.RegularExpressions;

   public class Tester
   {
      static void Main()
      {
         string s1 =
            "One, Two, Three Liberty Associates, Inc.";
         Regex theRegex = new Regex(" |, |,");
         StringBuilder sBuilder = new StringBuilder();
         int id = 1;

         foreach (string subString in theRegex.Split(s1))
         {
            sBuilder.AppendFormat(
               "{0}: {1}\n", id++, subString);
         }
         Console.WriteLine("{0}", sBuilder);
      }
   }
}
```

Вывод:

```
1: One
2: Two
3: Three
4: Liberty
5: Associates
6: Inc.
```

Пример 10.5 начинается с создания строки `s1`, похожей на ту, что была в примере 10.4:

```
string s1 = "One, Two, Three Liberty Associates, Inc.";
```

Далее создается регулярное выражение, которое будет использовано для поиска символов в строке:

```
Regex theRegex = new Regex(" |, |,");
```

Один из перегруженных конструкторов `Regex` принимает в качестве параметра строку, содержащую регулярное выражение. Это может вызвать некоторое непонимание. Что является регулярным выражением в программе на языке C# – текст, переданный конструктору, или собственно объект `Regex`? Справедливо утверждение, что строка, передаваемая конструктору, является регулярным выражением в традицион-

ном понимании этого термина. Однако с точки зрения объектно-ориентированного языка C# аргумент конструктора – это лишь строка символов, а объект theRegex как раз и есть регулярное выражение.

Остальная часть программы действует аналогично программе из примера 10.4, разве что вызывается не метод Split() строки s1, а Split() класса Regex. Метод Regex.Split() действует аналогично методу String.Split(), возвращая массив строк как результат разбора шаблона регулярного выражения из объекта theRegex.

Метод Regex.Split() является перегруженным. Простейшая его версия вызывается для экземпляра класса Regex, как показано в примере 10.5. Существует и статическая версия этого метода. Она принимает строку, в которой выполняется поиск, и шаблон поиска. Эта техника демонстрируется в примере 10.6.

Пример 10.6. Вызов статической версии Regex.Split( )

```
namespace Programming_CSharp
{
   using System;
   using System.Text;
   using System.Text.RegularExpressions;

   public class Tester
   {
      static void Main()
      {
         string s1 =
            "One, Two, Three Liberty Associates, Inc.";
         StringBuilder sBuilder = new StringBuilder();
         int id = 1;
         foreach (string subStr in Regex.Split(s1," |, |,"))
         {
            sBuilder.AppendFormat("{0}: {1}\n", id++, subStr);
         }
         Console.WriteLine("{0}", sBuilder);
      }
   }
}
```

Пример 10.6 идентичен примеру 10.5, но на этот раз объект типа Regex не создается. Вместо этого в примере 10.6 вызывается статическая версия метода Split(), принимающая два аргумента: строку, в которой выполняется поиск, и регулярное выражение, которое представляет шаблон для поиска соответствия.

Метод Split() экземпляра тоже перегружается версиями, ограничивающими количество разбиений и определяющими позицию в исходной строке, с которой следует начать поиск.

Класс коллекции Regex.MatchCollection

В пространстве имен RegularExpressions платформы .NET существуют два класса, позволяющие выполнять многократный поиск шаблона в строке и возвращать результаты в виде коллекции. Эта коллекция имеет тип MatchCollection и может содержать ноль и более объектов Match. Двумя важнейшими свойствами объекта Match являются его длина и значение. Любое из них может быть прочитано, как показано в примере 10.7.

Пример 10.7. Класс коллекции MatchCollection и объекты Match

```
namespace Programming_CSharp
{
   using System;
   using System.Text.RegularExpressions;

   class Test
   {
      public static void Main()
      {
         string string1 = "This is a test string";

         // найти символы, за которыми следует пробельный
         Regex theReg = new Regex(@"(\S+)\s");

         // получить коллекцию соответствующих фрагментов строк
         MatchCollection theMatches =
            theReg.Matches(string1);

         // перебрать члены коллекции
         foreach (Match theMatch in theMatches)
         {
            Console.WriteLine(
               "theMatch.Length: {0}", theMatch.Length);

            if (theMatch.Length != 0)
            {
               Console.WriteLine("theMatch: {0}",
                  theMatch.ToString());
            }
         }
      }
   }
}
```

Вывод:

```
theMatch.Length: 5
theMatch: This
theMatch.Length: 3
theMatch: is
theMatch.Length: 2
theMatch: a
theMatch.Length: 5
theMatch: test
```

В примере 10.7 создается простая строка, в которой будет выполнен поиск:

```
string string1 = "This is a test string";
```

и тривиальное регулярное выражение, определяющее шаблон поиска:

```
Regex theReg = new Regex(@"(\S+)\s");
```

Метасимволы `\S` определяют непробельный (обычный) символ, а знак «плюс» означает, что таких символов может быть один или более. Метасимволы `\s` (обратите внимание: в нижнем регистре) обозначают пробельный символ (символ-разделитель). В целом, шаблон определен как обычные символы, за которыми стоит символ-разделитель.

Вспомним, что символ «@» перед строкой делает ее дословным строковым литералом, избавляя программиста от необходимости дублировать символ обратной наклонной черты.

Выводимый программой текст свидетельствует о том, что найдены первые четыре слова. Заключительное слово найдено не было, поскольку за ним нет пробела. Если между словом `string` и закрывающими кавычками поставить пробел, программа найдет и это слово.

Свойство `Length` содержит длину обнаруженного фрагмента строки. Оно обсуждается в разделе «Коллекция CaptureCollection» далее в этой главе.

Группировка

Нередко бывает удобно сгруппировать выражения шаблонов таким образом, чтобы в исходной строке легче было выделить определенные фрагменты. Пусть, например, программисту требуется найти IP-адреса. Тогда он группирует все объекты `IPaddresses`, полученные из исходной строки.

IP-адрес используется для указания местоположения компьютера в сети и обычно имеет вид *x.x.x.x*, где *x* принимает значение от 0 до 255 (например, 192.168.0.1).

Класс `Group` позволяет группировать соответствия на основе синтаксиса регулярных выражений и представляет результаты действия одного группирующего выражения.

Группирующее выражение именует группу и предоставляет регулярное выражение. Любой фрагмент строки, удовлетворяющий этому регулярному выражению, будет добавлен в группу. Например, группа `ip` задается следующим выражением:

```
@"(?<ip>(\d|\.)+)\s"
```

Класс Match является производным от класса Group и имеет коллекцию Groups, которая содержит все группы, обнаруженные объектом Match.

Создание и использование коллекции Groups и классов Group иллюстрируется примером 10.8.

Пример 10.8. Класс Group

```
namespace Programming_CSharp
{
    using System;
    using System.Text.RegularExpressions;

    class Test
    {
        public static void Main()
        {
            string string1 = "04:03:27 127.0.0.0 LibertyAssociates.com";

            // группа time: одна или несколько цифр или
            // двоеточий, за которыми следует пробел
            Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
            // группа ip: одна или несколько цифр или
            // точек, за которыми следует пробел
            @"(?<ip>(\d|\.)+)\s" +
            // группа site: один или несколько символов
            @"(?<site>\S+)");

            // получить коллекцию соответствующих фрагментов строк
            MatchCollection theMatches = theReg.Matches(string1);

            // перебрать члены коллекции
            foreach (Match theMatch in theMatches)
            {
                if (theMatch.Length != 0)
                {
                  Console.WriteLine("\ntheMatch: {0}", theMatch.ToString());
                   Console.WriteLine("time: {0}", theMatch.Groups["time"]);
                   Console.WriteLine("ip: {0}", theMatch.Groups["ip"]);
                   Console.WriteLine("site: {0}", theMatch.Groups["site"]);
                }
            }
        }
    }
}
```

Пример 10.8 тоже начинается с создания строки, в которой выполняется поиск:

```
string string1 = "04:03:27 127.0.0.0 LibertyAssociates.com";
```

Такая строка может быть одной из большого количества записей в журнале веб-сервера или, например, результатом поиска в базе данных. В этом простом примере строка содержит три поля, разделенных

пробелами: время занесения записи в журнал, IP-адрес и сайт. Конечно, в реальной ситуации может потребоваться более сложный поиск с другой системой разделителей.

В примере 10.8 создается объект Regex, используемый для поиска в исходной строке и разбиения ее на три группы: time, ip и site. Строка с регулярным выражением достаточно проста, и разобраться в примере несложно. (Впрочем, следует помнить, что в реальной ситуации, скорее всего, выполнялся бы поиск лишь в какой-то части исходной строки, а не во всей строке, как в данном примере.)

```
// группа time: одна или несколько цифр или
// двоеточий, за которыми следует пробел
Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
// группа ip: одна или несколько цифр или
// точек, за которыми следует пробел
@"(?<ip>(\d|\.)+)\s" +
// группа site: один или несколько символов
@"(?<site>\S+)");
```

Сосредоточим внимание на символах, определяющих группу:

```
(?<time>
```

Группа создается круглыми скобками. Все, что находится между открывающей скобкой (непосредственно перед вопросительным знаком) и закрывающей (в данном случае – после плюса), является одиночной безымянной группой:

```
(@"(?<time>(\d|\:)+)
```

Последовательность символов ?<time> дает группе имя time и связывает ее с шаблоном, то есть с регулярным выражением ((\d|\:)+)\s". Это регулярное выражение может быть озвучено следующим образом: «одна или несколько цифр или двоеточий, за которыми следует пробел».

Аналогично строка ?<ip> именует группу ip, а строка ?<site> – группу site. Как и в примере 10.7, в примере 10.8 запрашивается коллекция всех найденных соответствий:

```
MatchCollection theMatches = theReg.Matches(string1);
```

Затем код перебирает элементы коллекции theMatches, выделяя каждый объект класса Match.

Если свойство Length очередного члена коллекции theMatches больше 0, значит, объект класса Match был найден. Тогда он выводится на экран.

```
Console.WriteLine("\ntheMatch: {0}", theMatch.ToString());
```

Выводимая строка выглядит так:

```
theMatch: 04:03:27 127.0.0.0 LibertyAssociates.com
```

Далее программа извлекает группу time из коллекции Groups объекта класса Match и выводит полученное значение:

```
Console.WriteLine("time: {0}", theMatch.Groups["time"]);
```

Выводится следующая строка:

```
time: 04:03:27
```

Аналогично получаются и выводятся группы ip и site:

```
Console.WriteLine("ip: {0}", theMatch.Groups["ip"]);
Console.WriteLine("site: {0}", theMatch.Groups["site"]);
```

Вывод:

```
ip: 127.0.0.0
site: LibertyAssociates.com
```

В примере 10.8 коллекция theMatches содержит только один объект Match. Однако вполне возможно искать в строке соответствия несколько выражений.

Чтобы убедиться в этом, измените строку string1 в примере 10.8 так, чтобы она содержала несколько записей в регистрационном журнале, а не одну:

```
string string1 = "04:03:27 127.0.0.0 LibertyAssociates.com " +
"04:03:28 127.0.0.0 foo.com " +
"04:03:29 127.0.0.0 bar.com " ;
```

Тогда в коллекции theMatches типа MatchCollection появятся три объекта и будет выведен следующий текст:

```
theMatch: 04:03:27 127.0.0.0 LibertyAssociates.com
time: 04:03:27
ip: 127.0.0.0
site: LibertyAssociates.com

theMatch: 04:03:28 127.0.0.0 foo.com
time: 04:03:28
ip: 127.0.0.0
site: foo.com

theMatch: 04:03:29 127.0.0.0 bar.com
time: 04:03:29
ip: 127.0.0.0
site: bar.com
```

В данном примере коллекция theMatches содержит три объекта типа Match. При каждом проходе внешнего цикла foreach из коллекции считывается очередной объект класса Match, и его содержимое выводится на экран:

```
foreach (Match theMatch in theMatches)
```

Для каждого из найденных объектов Match можно вывести его значение целиком, отдельные группы или и то и другое.

Класс коллекции CaptureCollection

Каждый раз, когда объект Regex соответствует одному из выражений, создается экземпляр класса Capture, который добавляется в коллекцию CaptureCollection. Каждый такой объект представляет собой отдельный результат поиска соответствия. У каждой группы есть коллекция таких результатов для своих выражений, связанная с данной группой.

Самым важным свойством объекта класса Capture является Length, то есть длина найденного фрагмента строки. Когда запрашивается длина объекта Match, возвращается именно Capture.Length, поскольку Match произведен от Group, а класс Group, в свою очередь, произведен от Capture.

Схема наследования регулярных выражений в .NET позволяет классу Match включать в свой интерфейс методы и свойства своих базовых классов. В определенном смысле Group является результатом поиска – это результат, инкапсулирующий идею группирования подвыражений. В свою очередь, Match является классом Group, будучи инкапсуляцией всех групп подвыражений, образующих результат поиска соответствия регулярному выражению в целом. (Более подробные рассуждения об отношении «является», а также о других отношениях приведены в главе 5.)

Самым вероятным случаем является наличие одного объекта Capture в коллекции CaptureCollection, но это совсем не обязательно. Рассмотрим, что получится, если в строке, в которой ведется поиск, название компании может встретиться в одной из двух позиций. Чтобы сгруппировать их в одной результирующей строке, нужно создать группу ?<company> в двух местах регулярного выражения:

```
Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
@"(?<company>\S+)\s" +
@"(?<ip>(\d|\.)+)\s" +
@"(?<company>\S+)\s");
```

Эта группа будет выбирать любую строку символов, которая следует за группой time, а также любую строку, следующую за группой ip. Имея такое регулярное выражение, можно смело приступать к поиску в строке:

```
string string1 = "04:03:27 Jesse 0.0.0.127 Liberty ";
```

Строка содержит имена в обеих указанных позициях, однако на экран выводится:

```
theMatch: 04:03:27 Jesse 0.0.0.127 Liberty
```

```
time: 04:03:27
ip: 0.0.0.127
Company: Liberty
```

Что случилось? Почему в группе `Company` **только строка** `Liberty`**? Где первое имя, которое тоже соответствует подвыражению? Объяснение состоит в том, что второе имя «затерло» собой первое. Однако группа нашла оба имени, о чем свидетельствует содержимое коллекции** `Captures` **из примера 10.9.**

Пример 10.9. Пример использования коллекции

```
namespace Programming_CSharp
{
   using System;
   using System.Text.RegularExpressions;

   class Test
   {
      public static void Main()
      {
         // строка для поиска
         // обратите внимание, что имена появляются
         // в обеих позициях
         string string1 = "04:03:27 Jesse 0.0.0.127 Liberty ";

         // регулярное выражение, группирующее company дважды
         Regex theReg = new Regex(@"(?<time>(\d|\:)+)\s" +
            @"(?<company>\S+)\s" +
            @"(?<ip>(\d|\.)+)\s" +
            @"(?<company>\S+)\s");

         // получить коллекцию соответствующих фрагментов строки
         MatchCollection theMatches =
            theReg.Matches(string1);

         // просмотреть члены коллекции
         foreach (Match theMatch in theMatches)
         {
            if (theMatch.Length != 0)
            {
               Console.WriteLine("theMatch: {0}", theMatch.ToString());
               Console.WriteLine("time: {0}", theMatch.Groups["time"]);
               Console.WriteLine("ip: {0}", theMatch.Groups["ip"]);
              Console.WriteLine("Company: {0}", theMatch.Groups["company"]);

               // просмотреть члены коллекции Captures
               // в группе company в коллекции Groups
               // для найденной подстроки
               foreach (Capture cap in
                  theMatch.Groups["company"].Captures)
               {
                  Console.WriteLine("cap: {0}",cap.ToString());
               }
```

```
            }
        }
    }
  }
}
```

Вывод:

```
theMatch: 04:03:27 Jesse 0.0.0.127 Liberty
time: 04:03:27
ip: 0.0.0.127
Company: Liberty
cap: Jesse
cap: Liberty
```

Фрагмент программы, выделенный полужирным шрифтом, выполняет циклический просмотр членов коллекции `Captures` для группы `Company`.

```
foreach (Capture cap in theMatch.Groups["company"].Captures)
```

Рассмотрим, как компилятор преобразовал эту строку программы. Вначале он нашел коллекцию, члены которой будут просматриваться. Переменная `theMatch` является объектом, у которого есть коллекция по имени `Groups`. Эта коллекция имеет индексатор, принимающий строку и возвращающий одиночный объект `Group`. Так, следующая строка возвращает объект `Group`:

```
theMatch.Groups["company"]
```

У этого объекта есть коллекция `Captures`. Следовательно, приведенная ниже строчка программы возвращает коллекцию `Capture` для объекта `Group`, сохраненного как `Groups["company"]` в объекте `theMatch`:

```
theMatch.Groups["company"].Captures
```

Цикл `foreach` просматривает все члены коллекции `Captures`, поочередно присваивая их локальной переменной `cap`, имеющей тип `Capture`. Из выводимого программой текста ясно, что коллекция содержит два элемента, `Jesse` и `Liberty`. Второй замещает первый, поэтому на экран выводится только `Liberty`. Тем не менее, изучение коллекции `Captures` позволяет обнаружить там оба найденных значения.

11

Обработка исключений

Язык C#, как и многие другие объектно-ориентированные языки, реагирует на ошибки и ненормальные ситуации с помощью вызова *исключений* (*exceptions*). Исключение – это объект, инкапсулирующий информацию о необычном программном происшествии.

Важно проводить различие между ошибкой в программе, ошибочной ситуацией и исключением. *Ошибка в программе* (*bug*) допущена программистом, и он должен исправить ее до передачи кода заказчику. Исключения не являются защитой от ошибок в программе. Хотя последние и могут быть причиной вызова исключения, не стоит рассчитывать, что исключения помогут обнаружить ошибки. Вместо этого следует отладить программу.

Ошибочная ситуация (*error*) вызвана действиями пользователя. Например, пользователь может ввести число вместо буквы. Такая ошибка тоже способна вызывать исключение, но программист в состоянии предотвратить это с помощью операторов, проверяющих допустимость поступающих данных. Следует предвидеть ошибочные ситуации и защищаться от них где только возможно.

Даже если программист исправил все свои ошибки и предвидит все пользовательские, он все равно будет сталкиваться с непредсказуемыми и неотвратимыми проблемами, такими как нехватка доступной памяти или попытка открыть несуществующий файл. Исключения нельзя предотвратить, но на них можно отреагировать так, что они не приведут к краху программы.

Когда программа сталкивается с исключительной ситуацией, например с нехваткой памяти, она *вызывает* (*throws*) исключение. После вызова исключения выполнение текущей функции прерывается и стек освобождается, пока не будет найден подходящий обработчик исключения.

Сказанное выше означает, что если текущая функция не обрабатывает исключение, то она заканчивается, а вызвавшая ее функция получает шанс обработать вызванное исключение. Если ни одна из функций в цепочке вызовов так и не обработает исключение, оно, в конце концов, попадет к CLR, и выполнение программы будет прекращено.

Обработчик исключения (*exception handler*) – это программный блок, предназначенный для реагирования на вызов исключения. Обработчики исключений реализуются в операторах `catch`. В идеальном случае, если исключение перехвачено и обработано, программа справляется с проблемой и продолжает свою работу. Даже если она будет не в состоянии продолжить выполнение, перехват исключений позволяет программисту как минимум выдать на экран осмысленное сообщение и закончить работу программы элегантно.

Если в тексте функции есть фрагмент, который должен быть выполнен независимо от того, было ли вызвано какое-либо исключение (например, фрагмент, освобождающий выделенные программе ресурсы), то его можно поместить в блок `finally`, где этот фрагмент программы наверняка будет выполнен даже после вызова исключения.

Вызов и обработка исключений

В языке C# в качестве исключений могут выступать только объекты типа `System.Exception` или объекты производного от него типа. Пространство имен `System` среды CLR включает в себя целый ряд типов исключений, которыми можно пользоваться в программе. Среди этих типов присутствуют `ArgumentNullException`, `InvalidCastException` и `OverflowException`, а также многие другие.

Оператор throw

Чтобы проинформировать о ненормальной ситуации, возникшей в каком-либо классе C#, программист вызывает исключение, пользуясь ключевым словом `throw`. В следующей строчке кода создается и вызывается новый объект класса `System.Exception`:

```
throw new System.Exception();
```

Вызов исключения сразу же прекращает выполнение программы, а среда CLR тем временем ищет процедуру обработки исключения. Если обработчик исключения отсутствует в текущем методе, стек вызовов освобождается, то есть с него удаляются вызвавшие методы, пока обработчик не будет найден. Если CLR пройдет стек вплоть до `Main()`, не найдя обработчик вызванного исключения, выполнение программы завершается. Этот процесс проиллюстрирован в примере 11.1.

Пример 11.1. Вызов исключения

```
namespace Programming_CSharp
{
```

```
    using System;

    public class Test
    {
        public static void Main()
        {
            Console.WriteLine("Вход в Main...");
            Test t = new Test();
            t.Func1();
            Console.WriteLine("Выход из Main...");

        }

        public void Func1()
        {
            Console.WriteLine("Вход в Func1...");
            Func2();
            Console.WriteLine("Выход из Func1...");
        }

        public void Func2()
        {
            Console.WriteLine("Вход в Func2...");
            throw new System.Exception();
            Console.WriteLine("Выход из Func2...");
        }
    }
}
```

Вывод:

```
Вход в Main...
Вход в Func1...
Вход в Func2...

Exception occurred: System.Exception: An exception of type
System.Exception was thrown.
   at Programming_CSharp.Test.Func2()
        in ...exceptions01.cs:line 26
   at Programming_CSharp.Test.Func1()
        in ...exceptions01.cs:line 20
   at Programming_CSharp.Test.Main()
        in ...exceptions01.cs:line 12
```

Эта простая программа выводит на экран сообщение о вызове каждого метода и выходе из него. Метод `Main()` создает экземпляр класса `Test` и вызывает метод `Func1()`. Выведя сообщение «Вход в Func1...», этот метод сразу вызывает метод `Func2()`. Тот, в свою очередь, выводит сообщение и вызывает объект типа `System.Exception`.

Выполнение программы останавливается, а среда CLR ищет обработчик исключения в методе `Func2()`. Не обнаружив его, CLR освобождает стек (так и не выведя сообщение «Выход из Func2...») вплоть до метода `Func1()`. Снова не найдя обработчик, CLR освобождает стек до метода `Main()`. Поскольку обработчика исключения нет и там, вызывается обработчик по умолчанию, который выводит сообщение об ошибке.

Оператор catch

В языке C# обработчик исключения называется *блоком catch*, а создается он с помощью ключевого слова `catch`.

В примере 11.2 оператор `throw` выполняется в блоке `try`, а блок `catch` служит для сообщения о том, что исключение перехвачено и обработано.

Пример 11.2. Обработка исключения

```
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        public static void Main()
        {
            Console.WriteLine("Вход в Main...");
            Test t = new Test();
            t.Func1();
            Console.WriteLine("Выход из Main...");
        }

        public void Func1()
        {
            Console.WriteLine("Вход в Func1...");
            Func2();
            Console.WriteLine("Выход из Func1...");
        }

        public void Func2()
        {
            Console.WriteLine("Вход в Func2...");
            try
            {
                Console.WriteLine("Вход в блок try ...");
                throw new System.Exception();
                Console.WriteLine("Выход из блока try ...");
            }
            catch
            {
                Console.WriteLine("Исключение перехвачено и обработано.");
            }
            Console.WriteLine("Выход из Func2...");
        }
    }
}
```

Вывод:

```
Вход в Main...
Вход в Func1...
Вход в Func2...
Вход в блок try...
```

```
Исключение перехвачено и обработано.
Выход из Func2...
Выход из Func1...
Выход из Main...
```

Пример 11.2 идентичен примеру 11.1 с тем отличием, что теперь программа содержит блок try/catch. Как правило, блок try содержит потенциально опасные действия, такие как обращение к файлу, выделение памяти и т. д.

Вслед за оператором try идет неспециализированный оператор catch. В данном примере оператор catch неспециализированный, поскольку не указано, какой тип исключений он должен обрабатывать, и поэтому он перехватит любое вызванное исключение. Применение операторов catch для обработки исключений конкретного типа обсуждается далее в этой главе.

Корректирующие действия

В примере 11.2 оператор catch просто сообщает о перехвате и обработке исключения. В реальной же ситуации, скорее всего, были бы предприняты действия по устранению проблемы, приведшей к вызову исключения. Например, если пользователь попытается открыть файл, предназначенный только для чтения, можно предоставить ему метод, позволяющий изменять атрибуты файла. Если программе не хватает памяти, можно дать пользователю возможность закрыть другие приложения. Если же справиться с ситуацией не удается, блок catch может хотя бы вывести сообщение, чтобы уведомить о возникшей проблеме.

Освобождение стека вызовов

Изучим результат выполнения программы, приведенной в примере 11.2, более внимательно. Видно, что управление передавалось методу Main(), методу Func1(), методу Func2() и, наконец, блоку try. Сообщение о выходе из блока try отсутствует, хотя блоки Func2(), Func1() и Main() завершили свою работу естественным образом. Что случилось?

Когда вызывается исключение, выполнение программы немедленно останавливается, а управление передается блоку catch. Этот блок *никогда* не возвращает управление в точку, где было остановлено выполнение. Поэтому сообщение о выходе из блока try никогда не будет выведено. Блок catch обрабатывает исключение, и выполнение программы продолжается с оператора, следующего за этим блоком.

В отсутствие оператора catch стек вызовов освобождается. Сейчас оператор catch расположен в том же методе, в котором было вызвано исключение, и вызов исключения не приводит к освобождению стека. Оно обрабатывается, проблемы снимаются, и программа продолжает свою работу. Чтобы все было до конца понятно, перенесем блоки try/catch в метод Func1(), как это сделано в примере 11.3.

Пример 11.3. Блок catch в вызывающей функции

```
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        public static void Main()
        {
            Console.WriteLine("Вход в Main...");
            Test t = new Test();
            t.Func1();
            Console.WriteLine("выход из Main...");

        }

        public void Func1()
        {
            Console.WriteLine("Вход в Func1...");
            try
            {
                Console.WriteLine("Вход в блок try ...");
                Func2();
                Console.WriteLine("Выход из блока try...");
            }
            catch
            {
                Console.WriteLine(
                   "Исключение перехвачено и обработано.");
            }

            Console.WriteLine("Выход из Func1...");
        }

        public void Func2()
        {
            Console.WriteLine("Вход в Func2...");
            throw new System.Exception();
            Console.WriteLine("Выход из Func2...");
        }
    }
}
```

Вывод:

```
Вход в Main...
Вход в Func1...
Вход в блок try...
Вход в Func2...
Исключение перехвачено и обработано.
Выход из Func1...
Выход из Main...
```

На этот раз исключение обрабатывается не в Func2(), а в Func1(). Когда вызывается метод Func2(), он выводит сообщение «Вход в Func2...» и вызывает исключение. Выполнение прекращается, CLR ищет обработчик исключения и не находит его. Тогда текущий метод удаляется из стека, а CLR ищет обработчик в методе Func1(). Управление передается в блок catch, и выполнение метода продолжается с оператора, следующего за этим блоком. Выводятся сообщения о выходе из Func1(), а затем из Main().

В настоящий момент очень важно, чтобы читатель полностью понимал, почему не выводятся сообщения о выходе из блока try и метода Func2() («Выход из блока Try...» и «Выход из Func2...» соответственно). Вот классический случай, когда пошаговое выполнение кода в отладчике проясняет самые сложные вопросы.

Создание специализированных операторов catch

До сих пор в коде примеров присутствовали только неспециализированные операторы catch. В то же время, можно написать такой оператор catch, который будет обрабатывать лишь некоторые исключения, принимая во внимание их тип. В примере 11.4 демонстрируется, как указать оператору catch, какое исключение следует перехватить и обработать.

Пример 11.4. Указание типа обрабатываемого исключения

```
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        public static void Main()
        {
            Test t = new Test();
            t.TestFunc();
        }

        // деление двух чисел с обработкой
        // вызываемых исключений
        public void TestFunc()
        {
            try
            {
                double a = 5;
                double b = 0;
                Console.WriteLine ("{0} / {1} = {2}",
                    a, b, DoDivide(a,b));
            }

            // первым идет "самый младший" из производных типов
            catch (System.DivideByZeroException)
            {
```

```
                Console.WriteLine(
                   "DivideByZeroException перехвачено!");
            }

            catch (System.ArithmeticException)
            {
                Console.WriteLine(
                   "ArithmeticException перехвачено!");
            }

            // последним идет неспецифицированный оператор catch
            catch
            {
                Console.WriteLine(
                   "Перехвачено неопознанное исключение");
            }

        }

        // выполнить деление, если оно корректно
        public double DoDivide(double a, double b)
        {
            if (b == 0)
                throw new System.DivideByZeroException();
            if (a == 0)
                throw new System.ArithmeticException();
            return a/b;
        }
    }
}
```

Вывод:

```
DivideByZeroException перехвачено!
```

В этом примере метод `DoDivide()` не позволит выполнить деление на ноль; впрочем, он не позволит и ноль разделить на какое-либо число. В первом случае (попытка деления на 0) вызывается объект `DivideByZeroException`. Если же попытаться разделить 0 на другое число, то подходящее исключение отсутствует, ведь деление нуля является допустимой математической операцией, которая не должна вызывать исключение. Просто в демонстрационных целях предположим, что деление нуля нежелательно, и вызовем исключение `ArithmeticException`.

Когда исключение вызвано, среда времени выполнения просматривает все процедуры обработки исключений *в порядке их следования* и передает управление первой подходящей. Если в программе установить значения `a=5` и `b=7`, то в результате ее выполнения будет выведена строка:

```
5 / 7 = 0.7142857142857143
```

Как и следовало ожидать, никакие исключения не вызываются. Если же изменить значение `a` на `0`, то будет выведено:

```
ArithmeticException перехвачено!
```

После вызова исключения CLR проверяет первый блок `catch`. Поскольку его тип аргумента `DivideByZeroException` не соответствует типу вызванного исключения, CLR переходит к следующей процедуре, обрабатывающей исключения `ArithmeticException`, и передает ей управление.

В заключительной проверке программы изменим значение `a` на `7`, а значение `b` на `0`. Будет вызвано исключение `DivideByZeroException`.

Программист должен быть очень внимательным в отношении порядка следования операторов `catch`. В рассмотренном примере тип `DivideByZeroException` является производным от `ArithmeticException`. Если поменять операторы `catch` местами, то исключение `DivideByZeroException` будет перехвачено обработчиком `ArithmeticException`, так и не дойдя до обработчика `DivideByZeroException`. И вообще, при таком порядке никакое исключение не достигнет обработчика `DivideByZeroException`. Компилятор распознает такую ситуацию и выдаст сообщение об ошибке!

Операторы `try/catch` можно распределить в программе так, что специализированные исключения будут обрабатываться в одном методе, а более общие – на более высоком уровне, то есть в вызывающей функции. Конкретное решение должно зависеть от задач, стоящих перед разработчиком.

Предположим, метод A вызывает метод B, который, в свою очередь, вызывает метод C. Метод C вызывает метод D, а тот – метод E. Выше всех расположены методы A и B; метод E – глубже всех. Если программист предвидит, что в методе E может быть вызвано исключение, он должен поместить блок `try/catch` вглубь кода, поближе к тому месту, где вероятны проблемы. При желании можно расположить более общие обработчики исключений на более высоких уровнях, на случай возникновения непредвиденных исключений.

Оператор finally

В некоторых ситуациях вызов исключения и освобождение стека вызовов могут создать определенные проблемы. Например, если программа открыла файл или как-то иначе занимает ресурс, то она должна получить возможность закрыть файл или сохранить содержимое буфера.

В C# это не такая серьезная проблема, как в других языках, например в C++. Сборка мусора не позволяет исключению стать причиной утечки памяти.

Итак, если некоторое действие (например закрытие файла) следует предпринять независимо от того, было ли вызвано исключение, у про-

граммиста есть выбор из двух стратегий. Первый подход состоит в размещении потенциально опасного кода в блоке try и вызове метода для закрытия файла как в блоке try, так и в блоке catch. Однако это решение не элегантно, поскольку включает в себя чреватое ошибками дублирование кода. Язык C# предлагает более удачную альтернативу в виде блока finally.

Код в блоке finally гарантированно выполняется независимо от того, было ли вызвано исключение. Метод TestFunc() имитирует открытие файла в начале своей работы. Затем он выполняет некоторые математические действия, после чего закрывает файл. Иногда в промежутке между открытием и закрытием файла будет вызываться исключение. Если оно будет вызвано, файл можно спокойно оставить открытым. Разработчик уверен – что бы ни случилось, в заключительной части метода файл будет закрыт. Действия по закрытию файла перенесены в блок finally, где они будут выполнены, невзирая на исключения.

Пример 11.5. Использование блока finally

```
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        public static void Main()
        {
            Test t = new Test();
            t.TestFunc();
        }

        // деление двух чисел с обработкой
        // вызываемых исключений
        public void TestFunc()

        {
            try
            {
                Console.WriteLine("Открытие файла");
                double a = 5;
                double b = 0;
                Console.WriteLine ("{0} / {1} = {2}",
                    a, b, DoDivide(a,b));
                Console.WriteLine (
                    "Эта строка может и не печататься");
            }

            // первым идет "самый младший" из производных типов
            catch (System.DivideByZeroException)
            {
                Console.WriteLine(
                   "DivideByZeroException перехвачено!");
            }
```

```
            catch
            {
                Console.WriteLine("Перехвачено неопознаное исключение");
            }
            finally
            {
                Console.WriteLine ("Закрытие файла.");
            }

        }

        // выполнить деление, если оно корректно
        public double DoDivide(double a, double b)
        {
            if (b == 0)
                throw new System.DivideByZeroException();
            if (a == 0)
                throw new System.ArithmeticException();
            return a/b;
        }
    }
}
```

Вывод:

```
Открытие файла
DivideByZeroException перехвачено!
Закрытие файла.
```

Вывод, когда b = 12:

```
Открытие файла
5 / 12 = 0.416666666666667

Эта строка может и не печататься
Закрытие файла.
```

В этом примере ради экономии места убран один из блоков `catch`, зато добавлен блок `finally`. Блок `finally` будет выполнен вне зависимости от того, было ли вызвано исключение. Поэтому фраза «Закрытие файла» присутствует в выводимом тексте в обоих случаях.

Блок `finally` не требует присутствия блоков `catch`, однако он требует обязательного присутствия блока `try`. Нельзя выходить из блока `finally` с помощью операторов `break`, `continue`, `return` и `goto`.

Объекты Exception

До сих пор в этой главе исключения служили лишь индикатором ошибки, но сам объект `Exception` не изучался. В то же время, класс `System.Exception` обладает рядом полезных методов и свойств. В частности, свойство `Message` предоставляет информацию об исключении,

например о причинах его возникновения. Свойство Message доступно только для чтения, а программа, вызывающая исключение, может передать это свойство конструктору исключения в качестве аргумента.

Свойство HelpLink предоставляет ссылку на справочный файл, связанный с исключением. Это свойство доступно как для чтения, так и для записи.

Свойство StackTrace доступно только для чтения и устанавливается на этапе выполнения. В примере 11.6 устанавливается свойство Exception.HelpLink, которое затем считывается для предоставления пользователю информации об исключении DivideByZeroException. Свойство StackTrace этого исключения используется для отслеживания в стеке пути к оператору, вызвавшему ошибку. Путь в стеке – это содержимое *стека вызовов* (*call stack*), то есть цепочка вызовов, ведущая к оператору, для которого было вызвано исключение.

Пример 11.6. Работа с объектом Exception

```
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        public static void Main()
        {
            Test t = new Test();
            t.TestFunc();
        }

        // деление двух чисел с обработкой
        // вызываемых исключений
        public void TestFunc()
        {
            try
            {
                Console.WriteLine("Открытие файла");
                double a = 12;
                double b = 0;
                Console.WriteLine ("{0} / {1} = {2}",
                    a, b, DoDivide(a,b));
                Console.WriteLine (
                  "Эта строка может и не печататься");
            }

            // первым идет "самый младший" из производных типов
            catch (System.DivideByZeroException e)
            {
                Console.WriteLine(
                     "\nDivideByZeroException! Msg: {0}",
                     e.Message);
                Console.WriteLine(
```

```
                    "\nHelpLink: {0}", e.HelpLink);
                Console.WriteLine(
                    "\nТрассировка стека: {0}\n",
                    e.StackTrace);
            }
            catch
            {
                Console.WriteLine(
                    "Перехвачено неопознанное исключение");
            }
            finally
            {
                Console.WriteLine (
                  "Закрытие файла.");
            }

        }

        // выполнить деление, если оно корректно
        public double DoDivide(double a, double b)
        {
            if (b == 0)
            {
                DivideByZeroException e =
                  new DivideByZeroException();
                  e.HelpLink =
                    "http://www.libertyassociates.com";
                throw e;
            }
            if (a == 0)
                throw new ArithmeticException();
            return a/b;
        }
    }
}
```

Вывод:

```
Открытие файла

DivideByZeroException! Msg: Attempted to divide by zero.

HelpLink: http://www.libertyassociates.com

Трассировка стека:
at Programming_CSharp.Test.DoDivide(Double a, Double b)
 in c:\...exception06.cs:line 56
at Programming_CSharp.Test.TestFunc()
in...exception06.cs:line 22

Закрытие файла.
```

В тексте, выведенном этой программой, список методов представлен в порядке, обратном тому, в котором они вызывались. То есть ошибка произошла в методе `DoDivide()`, который был вызван методом `Test-`

Func(). Когда методы вложены достаточно глубоко, путь в стеке помогает понять, как они были вызваны.

В данном примере не просто вызывается исключение DivideByZeroException, а создается новый объект Exception:

```
DivideByZeroException e = new DivideByZeroException();
```

Поскольку никакое сообщение конструктору не передается, на экран будет выведено сообщение, установленное по умолчанию:

```
DivideByZeroException! Msg: Attempted to divide by zero.
```

Программист может передать конструктору свое сообщение:

```
new DivideByZeroException("Вы пытались делить на ноль, это бессмысленно");
```

Тогда вместо стандартного появится сообщение:

```
DivideByZeroException! Msg: Вы пытались делить на ноль, это бессмысленно
```

Перед вызовом исключения устанавливается свойство HelpLink:

```
e.HelpLink =  "http://www.libertyassociates.com";
```

При обработке данного исключения программа выведет на экран сообщение и значение свойства HelpLink:

```
catch (System.DivideByZeroException e)
{
    Console.WriteLine("\nDivideByZeroException! Msg: {0}",
        e.Message);
    Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
```

Это позволит предоставить пользователю необходимую информацию. Путь в стеке вызовов, точнее, свойство StackTrace, тоже выводится на экран:

```
Console.WriteLine("\nТрассировка стека: {0}\n", e.StackTrace);
```

Вывод этого метода отражает полный путь, ведущий к точке вызова исключения:

```
Трассировка стека:
at Programming_CSharp.Test.DoDivide(Double a, Double b)
 in c:\...exception06.cs:line 56
at Programming_CSharp.Test.TestFunc()
in...exception06.cs:line 22
```

Обратите внимание, что здесь пути к файлу сокращены; в действительности выводимый текст выглядит несколько иначе.

Вызов пользовательских исключений

В большинстве случаев программисту достаточно базового набора исключений, предоставляемых средой CLR, в сочетании с пользовательскими сообщениями, которые программист передает конструктору, как показано в предыдущем примере. По крайней мере, этой информации достаточно блоку catch, обрабатывающему вызванное исключение. Однако возникают ситуации, когда нужно выдать более подробную информацию или снабдить исключение специальными возможностями. Создание собственного класса *пользовательских исключений* – задача вполне тривиальная. Единственное требование, которое предъявляется к классу, состоит с том, что он должен быть потомком (непосредственным или косвенным) класса System.ApplicationException. Создание пользовательского исключения демонстрируется в примере 11.7.

Пример 11.7. Создание пользовательского исключения

```
namespace Programming_CSharp
{
   using System;

   public class MyCustomException : System.ApplicationException
   {
      public MyCustomException(string message):
         base(message)
      {

      }
   }

   public class Test
   {
      public static void Main()
      {
         Test t = new Test();
         t.TestFunc();
      }

        // деление двух чисел с обработкой
        // вызываемых исключений
      public void TestFunc()
      {
         try
         {
            Console.WriteLine("Открытие файла");
            double a = 0;
            double b = 5;
            Console.WriteLine ("{0} / {1} = {2}", a, b, DoDivide(a,b));
            Console.WriteLine ("Эта строка может и не печататься");
         }

            // первым идет "самый младший" из производных типов
```

```
            catch (System.DivideByZeroException e)
            {
               Console.WriteLine(
                  "\nDivideByZeroException! Msg: {0}", e.Message);
               Console.WriteLine("\nHelpLink: {0}\n", e.HelpLink);
            }
            catch (MyCustomException e)
            {
               Console.WriteLine("\nMyCustomException! Msg: {0}", e.Message);
               Console.WriteLine("\nHelpLink: {0}\n", e.HelpLink);
            }
            catch
            {
               Console.WriteLine("Перехвачено неопознанное исключение");
            }
            finally
            {
               Console.WriteLine ("Закрытие файла.");
            }

         }

         // выполнить деление, если оно корректно
         public double DoDivide(double a, double b)
         {
            if (b == 0)
            {
               DivideByZeroException e = new DivideByZeroException();
               e.HelpLink="http://www.libertyassociates.com";
               throw e;
            }
            if (a == 0)
            {
               MyCustomException e = new MyCustomException(
                              "Не могу делить ноль на число");
               e.HelpLink = "http://www.libertyassociates.com/
                              NoZeroDivident.htm";
               throw e;
            }
            return a/b;
         }
      }
   }
```

Класс `MyCustomException` является потомком класса `System.ApplicationException` и состоит лишь из конструктора, принимающего строку `message`, которую он передает базовому классу, как описано в главе 4. В данном примере достоинство класса пользовательского исключения состоит в том, что он лучше отражает идеологию класса `Test`, в котором запрещено делить ноль на другие числа. Вызов исключения `ArithmeticException` вполне допустим, но оно может сбить с толку других

программистов, поскольку использование ноля в качестве делимого обычно ошибкой не считается.

Повторный вызов исключения

Иногда бывает необходимо, чтобы блок catch, предприняв некоторые корректирующие действия, вызывал исключение для внешнего блока try (в вызвавшем методе). Это может быть *то же самое* исключение, но может быть и другое. В последнем случае было бы разумно вложить первое исключение во второе, чтобы вызвавший метод знал историю вызова исключений. Свойство InnerException второго исключения возвращает первое исключение.

Поскольку свойство InnerException само является объектом Exception, оно может иметь свое вложенное исключение. Таким образом, исключения могут быть вложены друг в друга, как матрешки. Пример 11.8 является иллюстрацией сказанного.

Пример 11.8. Повторный вызов и вложенные исключения

```
namespace Programming_CSharp
{
   using System;

   public class MyCustomException : System.Exception
   {
      public MyCustomException(
         string message,Exception inner):
         base(message,inner)
      {

      }
   }

   public class Test
   {
      public static void Main()
      {
         Test t = new Test();
         t.TestFunc();
      }

      public void TestFunc()
      {
         try
         {
            DangerousFunc1();
         }

         // при обработке пользовательского исключения
         // вывести историю исключений
         catch (MyCustomException e)
```

```
    {
       Console.WriteLine("\n{0}", e.Message);
       Console.WriteLine(
          "Получаем историю исключений...");
       Exception inner =
          e.InnerException;
       while (inner != null)
       {
          Console.WriteLine(
             "{0}",inner.Message);
          inner =
             inner.InnerException;
       }
    }
}

public void DangerousFunc1()
{
   try
   {
      DangerousFunc2();
   }

   // при обработке исключения
   // вызывать пользовательское исключение
   catch(System.Exception e)
   {
      MyCustomException ex =
         new MyCustomException(
            "E3 - Пользовательская исключительная ситуация!",e);
      throw ex;
   }
}

public void DangerousFunc2()
{
   try
   {
      DangerousFunc3();
   }

   // при перехвате исключения DivideByZeroException
   // предпринять корректирующие действия и
   // вызвать базовый тип исключения
   catch (System.DivideByZeroException e)
   {
      Exception ex =
         new Exception("E2 - Func2 перехвачено деление на ноль",e);
      throw ex;
   }
}

public void DangerousFunc3()
```

```
        {
            try
            {
                DangerousFunc4();
            }
            catch (System.ArithmeticException)
            {
                throw;
            }

            catch (System.Exception)
            {
                Console.WriteLine("Обработка исключения.");
            }
        }

        public void DangerousFunc4()
        {
            throw new DivideByZeroException("E1 - DivideByZero Exception");
        }
    }
}
```

Вывод:

```
E3 - Пользовательская исключительная ситуация!
Получаем историю исключений...
E2 - Func2 перехвачено деление на ноль
E1 - DivideByZeroException
```

Поскольку из этой программы удалено все лишнее, выводимый ею текст может несколько озадачить читателя. Чтобы разобраться в программе, лучше всего запустить ее в отладчике и выполнить в пошаговом режиме.

Работа программы начинается с вызова метода `DangerousFunc1()` в блоке `try`:

```
try
{
    DangerousFunc1();
}
```

Метод `DangerousFunc1()` вызывает метод `DangerousFunc2()`, а тот вызывает `DangerousFunc3()`, который, в свою очередь, вызывает `DangerousFunc4()`. Все эти вызовы помещены в блоки `try`. Метод `DangerousFunc4()` вызывает исключение `DivideByZeroException`. Вообще говоря, у исключения `System.DivideByZeroException` есть свое сообщение, но программист волен передать какое-нибудь другое. Здесь сообщение «E1 – DivideByZeroException» служит лишь для идентификации.

Исключение, вызванное методом `DangerousFunc4()`, обрабатывается в блоке `catch` метода `DangerousFunc3()`. Логика этого метода такова, что при обработке любого исключения типа `ArithmeticException` (например

исключения DivideByZeroException) он не предпринимает ничего, кроме повторного вызова исключения:

```
catch (System.ArithmeticException)
{
   throw;
}
```

Синтаксис повторного вызова того же самого исключения (без модификации) состоит лишь из слова throw.

Итак, исключение передано методу DangerousFunc2(), который обрабатывает его, предпринимает корректирующие действия и вызывает новое исключение типа Exception. Конструктору этого нового исключения метод DangerousFunc2() передает пользовательское сообщение «E2 – Func2 перехвачено деление на ноль» и *исходное исключение*. Таким образом, исходное исключение (E1) становится свойством InnerException для нового исключения (E2). Затем метод DangerousFunc2() вызывает новое исключение E2 для метода DangerousFunc1().

Метод DangerousFunc1() перехватывает это исключение, обрабатывает его, создает новое исключение типа MyCustomException (E3) и вкладывает в него исключение типа Exception (E2), в которое уже вложено исключение типа DivideByZeroException (E1). Все это передается функции TestFunc(), где и обрабатывается.

Когда выполняется оператор catch, он выводит сообщение:

```
E3 - Пользовательская исключительная ситуация!
```

после чего спускается вниз по вложенным исключениям, распечатывая их сообщения:

```
while (inner != null)
{
    Console.WriteLine("{0}",inner.Message);
    inner = inner.InnerException;
}
```

Текст, выводимый программой, регистрирует цепочку вызванных и обработанных исключений:

```
Получение истории исключений...
E2 - Func2 перехвачено деление на ноль
E1 - DivideByZero Exception
```

12

Делегаты и события

Когда умирает глава какого-нибудь государства, президент США не всегда имеет возможность присутствовать на похоронной церемонии. Вместо себя он посылает делегата. Часто таким делегатом является вице-президент, а если и его присутствие невозможно, президент посылает кого-нибудь другого, например государственного секретаря или даже первую леди. Президент не связывает такую ответственность с каким-то одним человеком; он делегирует свои полномочия тому, кто в состоянии соблюсти международный протокол.

Президент заранее определяет, какие функции должны быть выполнены (присутствие на похоронах), какие параметры следует передать (соболезнования) и что он рассчитывает услышать в ответ (слова признательности). Затем он наделяет конкретного человека делегируемыми полномочиями «на этапе выполнения», и это его, президента, рутинные обязанности.

В программировании нередко приходится сталкиваться с ситуациями, когда требуется выполнить некоторое действие, но заранее не известно, какой метод и даже какой объект будет выполнять его. Например, кнопка на экране знает, что она должна уведомить *некий* объект, когда пользователь щелкнет по ней. Однако она может и не знать, какому объекту или объектам нужно посылать уведомление. Вместо того чтобы жестко связывать кнопку с конкретным объектом, программист связывает ее с *делегатом*, а на этапе выполнения назначает делегатом тот метод, который потребуется.

В стародавние времена, на заре программирования, программа начинала свое выполнение, проделывала предопределенные шаги и завершала работу. Если в этот процесс включался пользователь, взаимодействие с ним было ограниченным и сводилось к заполнению полей.

В наши дни программная модель, основанная на графическом пользовательском интерфейсе, требует другого подхода, известного как *программирование, управляемое событиями*. Современная программа предоставляет пользователю интерфейс и ждет, когда он предпримет какое-либо действие. У пользователя богатый выбор таких действий. Он может выбирать команды меню, нажимать кнопки, вносить изменения в текстовые поля, щелкать по значкам и т. д. Каждое действие приводит к возникновению события. Кроме того, существуют события, непосредственно не связанные с действиями пользователя, например срабатывание таймера, приход сообщения по электронной почте или окончание операции копирования файлов.

Событие инкапсулирует идею «произошло нечто важное», и программа должна на него отреагировать. События и делегаты являются тесно связанными понятиями, поскольку гибкая обработка событий требует точного выбора обработчика. Обработчик события реализуется на языке C#, как правило, в виде делегата.

Делегаты используются и как процедуры обратного вызова, когда один класс как бы говорит другому: «Сделай эту работу, а когда закончишь – дай мне знать». Это второе применение делегатов подробно обсуждается в главе 21. Делегаты еще используются для указания методов, которые становятся известными только на этапе выполнения. Эта тема раскрывается в следующих разделах.

Делегаты

В языке C# делегаты – это полноценные объекты, без оговорок поддерживаемые языком. Технически делегат – это ссылочный тип, инкапсулирующий метод с указанной сигнатурой и возвращаемым типом. В делегате можно инкапсулировать любой подходящий метод. (В C++ и многих других языках программирования аналогичные цели достигаются с помощью указателей на методы класса и на функции. В отличие от них, делегаты объектно-ориентированы и обеспечивают проверку типов.)

Делегат создается ключевым словом `delegate`, за которым указывается возвращаемый тип и сигнатура делегируемых методов, например, так:

```
public delegate int WhichIsFirst(object obj1, object obj2);
```

В этом объявлении определяется делегат с именем `WhichIsFirst`, который инкапсулирует любой метод, принимающий два параметра типа `Object`, и возвращает значение целого типа.

Когда делегат определен, программист может инкапсулировать метод, создав экземпляр делегата путем передачи ему метода, соответствующего по сигнатуре и возвращаемому типу.

Применение делегатов для выбора методов на этапе выполнения

Делегаты указывают методы, которые могут быть использованы при обработке событий, а также для реализации обратных вызовов в программе. Кроме того, они применяются для указания статических методов и методов экземпляра, о которых ничего не известно до этапа выполнения.

Предположим, что программист хочет создать простой контейнерный класс `Pair`, который может принять и упорядочить любые два объекта. Невозможно знать заранее, какие именно объекты будут переданы, но можно внутри объектов определить методы сортировки и делегировать ответственность за упорядочивание объектов внутри класса самим этим объектам.

Разные объекты упорядочиваются по разным критериям. Например, два счетчика будут отсортированы по их числовым значениям, а две кнопки – в алфавитном порядке, по надписям на них. Разработчик класса `Pair` вправе ожидать, что объекты в паре знают, который из них должен быть первым. Поэтому он будет настаивать, чтобы объекты, передаваемые классу `Pair`, имели методы сортировки.

Программист определяет требуемый метод, создавая делегат с сигнатурой и возвращаемым типом метода, который должен быть предоставлен объектом (например кнопкой), чтобы класс `Pair` мог решить, какой объект первый, а какой – второй.

Класс `Pair` определяет делегат `WhichIsFirst`, а метод `Sort()` примет в качестве параметра экземпляр `WhichIsFirst`. Когда классу `Pair` потребуется узнать, в каком порядке хранить объекты, он вызовет делегат, передав ему два объекта в качестве аргументов. Ответственность за определение порядка следования объектов будет возложена на метод, инкапсулируемый делегатом.

Для тестирования делегата создадим два класса, `Dog` (Собака) и `Student` (Студент). У собак и студентов мало общего, но и те и другие знают, как решить, кто из них первый, то есть реализовать методы, которые можно инкапсулировать делегатом `WhichIsFirst`. Следовательно, как объекты класса `Dog`, так и объекты класса `Student` годятся для размещения в классе `Pair`.

В тестовой программе создадим пару объектов класса `Student` и столько же класса `Dog`, а затем сохраним их в объектах класса `Pair`. Создадим объекты делегатов, инкапсулирующие соответствующие методы, удовлетворяющие требованиям на сигнатуру и возвращаемый тип. Наконец, попросим объекты `Pair` отсортировать объекты `Dog` и `Student`. Ниже описывается, как все это реализуется в программе.

Вначале создадим конструктор Pair, который принимает два объекта и помещает их в закрытый массив:

```
public class Pair
{

    // два объекта, сохраняемые в порядке поступления
    public Pair(object firstObject, object secondObject)
    {
        thePair[0] = firstObject;
        thePair [1] = secondObject;
    }
    // массив для двух объектов
    private object[]thePair = new object[2];
```

Затем перегружается метод ToString(), возвращающий строковые значения двух объектов:

```
public override string ToString()
{
    return thePair [0].ToString() + ", " + thePair [1].ToString();
}
```

Итак, в объекте Pair находятся два объекта, и их значения можно вывести. Все готово к сортировке и выводу ее результатов. Поскольку тип объектов заранее неизвестен, ответственность за их упорядочивание в объекте Pair перекладывается на них самих. Поэтому от каждого объекта, хранящегося в Pair, требуется, чтобы он реализовал метод, возвращающий информацию, который из двух объектов первый. Такой метод будет принимать два объекта (произвольного типа) и возвращать перечислимое значение: theFirstComesFirst, если первым является первый параметр, и theSecondComesFirst, если первым должен быть второй.

Эти методы будут инкапсулированы делегатом WhichIsFirst, определенным в рамках класса Pair:

```
public delegate comparison
WhichIsFirst(object obj1, object obj2);
```

Возвращаемое значение имеет перечислимый тип comparison:

```
public enum comparison
{
   theFirstComesFirst = 1,
   theSecondComesFirst = 2
}
```

Любой статический метод, который принимает два объекта и возвращает значение типа comparison, может быть инкапсулирован этим делегатом на этапе выполнения.

Теперь для класса Pair можно определить метод Sort():

```
public void Sort(WhichIsFirst theDelegatedFunc)
{
```

```
        if (theDelegatedFunc(thePair[0],thePair[1]) ==
           comparison.theSecondComesFirst)
        {
           object temp = thePair[0];
           thePair[0] = thePair[1];
           thePair[1] = temp;
        }
    }
```

Этот метод имеет один параметр, делегат типа WhichIsFirst с именем theDelegatedFunc. Метод Sort() делегирует ответственность за упорядочивание двух объектов в объекте Pair методу, инкапсулированному этим делегатом. В теле метода Sort() вызывается делегированный метод и анализируется возвращенное им значение, которое является одним из двух перечисляемых значений типа comparsion.

Если возвращено значение theSecondComesFirst, объекты внутри Pair меняются местами; в противном случае ничего не предпринимается.

Обратите внимание, что theDelegatedFunc является именем параметра, представляющего метод, инкапсулируемый делегатом. В этом аргументе можно передать любой метод (с соответствующей сигнатурой и возвращаемым типом). Здесь полная аналогия с ситуацией, в которой некий метод принимает в качестве параметра целое число:

```
int SomeMethod (int myParam){//...}
```

В этой строке программы myParam – имя параметра, но методу SomeMethod можно передать любое значение типа int. В рассматриваемой программе имя параметра – theDelegatedFunc, но в качестве аргумента ему можно передать любой метод, удовлетворяющий требованиям делегата WhichIsFirst к сигнатуре и возвращаемому типу.

Будем сортировать студентов по их именам. Напишем метод, который возвращает значение theFirstComesFirst, если имя первого студента идет по алфавиту раньше имени второго, и значение theSecondComesFirst, если имя второго идет первым. Получив в качестве параметров «Amy, Beth», метод возвратит theFirstComesFirst, а если ему передать «Beth, Amy», он возвратит theSecondComesFirst. Когда метод Sort() получит значение theSecondComesFirst, он поменяет местами элементы массива, ставя Amy на первое место, а Beth на второе.

Теперь напишем еще один метод, ReverseSort(), который меняет порядок следования элементов массива на обратный:

```
public void ReverseSort(WhichIsFirst theDelegatedFunc)
{
    if (theDelegatedFunc(thePair[0], thePair[1]) ==
          comparison.theFirstComesFirst)
    {
        object temp = thePair[0];
        thePair[0] = thePair[1];
```

```
            thePair[1] = temp;
        }
    }
```

Логика этого метода идентична логике метода `Sort()`, но метод `ReverseSort()` меняет местами элементы массива, если делегированный метод сообщает, что первый элемент должен быть первым. Итак, если делегированному методу передать «Amy, Beth», то он возвратит `theFirstComesFirst` (то есть имя Amy идет раньше), а метод *обратной* сортировки поменяет имена местами, поставив Beth в начало. Подобный подход позволяет в методе `Sort()` использовать для прямой и обратной сортировки одну и ту же делегируемую функцию, не заставляя объект поддерживать отдельную функцию, которая возвращала бы наименьшее значение из двух.

Теперь нужны какие-нибудь объекты сортировки. Создадим два очень простых класса, `Student` и `Dog`. В момент создания присвоим объекту `Student` какое-либо имя:

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

Классу `Student` требуются два метода, один для перегрузки метода `ToString()`, а другой – для инкапсуляции в качестве делегируемого метода.

Класс `Student` должен перегрузить метод `ToString()` так, чтобы одноименный метод в классе `Pair` (вызывающий `ToString()` для объектов, хранящихся в `Pair`) работал корректно. Реализация метода в классе `Student` всего лишь возвращает имя студента, которое уже является строкой:

```
public override string ToString()
{
     return name;
}
```

Класс `Student` должен также реализовать метод, которому метод `Pair.Sort()` будет делегировать ответственность за определение порядка элементов:

```
return (String.Compare(s1.name, s2.name) < 0 ?
   comparison.theFirstComesFirst :
   comparison.theSecondComesFirst);
```

Здесь `String.Compare` является методом, предоставляемым платформой .NET Framework для класса `String`. Он сравнивает две строки и возвра-

щает отрицательное значение, если первая строка меньше, положительное значение, если она больше, и ноль, если строки одинаковы. Более подробно этот метод обсуждался в главе 10. Обратите внимание на логику приведенного выше фрагмента программы. Значение theFirstComesFirst возвращается, только если первая строка меньше; если они равны или первая больше, возвращается theSecondComesFirst.

Метод WhichStudentComesFirst() принимает в качестве параметров два объекта, а возвращает значение типа comparison. Он годится на роль делегируемого метода для Pair.WhichIsFirst, поскольку удовлетворяет требованиям, предъявляемым к сигнатуре и возвращаемому типу.

Второй класс называется Dog. Собак будем сортировать по весу; более легкую поставим перед более тяжелой. Вот полное определение класса Dog:

```
public class Dog
 {
    public Dog(int weight)
    {
        this.weight=weight;
    }

      // собаки упорядочиваются по весу
      public static comparison WhichDogComesFirst(
         Object o1, Object o2)
      {
         Dog d1 = (Dog) o1;
         Dog d2 = (Dog) o2;
        return d1.weight > d2.weight ? theSecondComesFirst :
                                       theFirstComesFirst;
      }
    public override string ToString()
    {
        return weight.ToString();
    }
    private int weight;
}
```

Обратите внимание, что в классе Dog тоже перегружается метод ToString() и реализуется статический метод с соответствующей сигнатурой для делегирования ему полномочий. Кроме того, отметим, что у делегируемых методов классов Dog и Student разные имена. Они вовсе не обязаны носить одно и то же имя, поскольку назначаются делегатами динамически, на этапе выполнения.

Делегируемым методам можно давать какие угодно имена. Однако если придерживаться определенной системы (например WhichDogComesFirst и WhichStudentComesFirst), то код будет удобочитаемым, понятным и простым в сопровождении.

Пример 12.1 представляет собой законченную программу, демонстрирующую вызовы делегированных методов.

Пример 12.1. Работа с делегатами

```
namespace Programming_CSharp
{
   using System;

   public enum comparison
   {
      theFirstComesFirst = 1,
      theSecondComesFirst = 2
   }

// простая коллекция из двух элементов
   public class Pair
   {
      // объявление делегата
      public delegate comparison WhichIsFirst(object obj1, object obj2);

      // конструктор принимает два объекта и сохраняет их в порядке
      // поступления
      public Pair(
         object firstObject,
         object secondObject)
      {
         thePair[0] = firstObject;
         thePair[1] = secondObject;
      }

      // открытый метод, упорядочивающий два объекта
      // по определяемому ими критерию
      public void Sort(
         WhichIsFirst theDelegatedFunc)
      {
         if (theDelegatedFunc(thePair[0],thePair[1])
            == comparison.theSecondComesFirst)
         {
            object temp = thePair[0];
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
      }

      // открытый метод, расставляющий два объекта по порядку, обратному
      // определяемому ими самими критерию
      public void ReverseSort(
         WhichIsFirst theDelegatedFunc)
      {
         if (theDelegatedFunc(thePair[0],thePair[1]) ==
            comparison.theFirstComesFirst)
         {
            object temp = thePair[0];
```

```
            thePair[0] = thePair[1];
            thePair[1] = temp;
         }
      }

      // запросить у двух объектов их строковые значения
      public override string ToString(  )
      {
         return thePair[0].ToString(  ) + ", "
            + thePair[1].ToString(  );
      }

      // закрытый массив для хранения двух объектов
      private object[] thePair = new object[2];
   }

   public class Dog
   {
      public Dog(int weight)
      {
         this.weight=weight;
      }

      // собаки упорядочиваются по весу
      public static comparison WhichDogComesFirst(
         Object o1, Object o2)
      {
         Dog d1 = (Dog) o1;
         Dog d2 = (Dog) o2;
         return d1.weight > d2.weight ?
            comparison.theSecondComesFirst :
            comparison.theFirstComesFirst;
      }
      public override string ToString(  )
      {
         return weight.ToString(  );
      }
      private int weight;
   }

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

      // студенты упорядочиваются по алфавиту
      public static comparison
         WhichStudentComesFirst(Object o1, Object o2)
      {
         Student s1 = (Student) o1;
         Student s2 = (Student) o2;
         return (String.Compare(s1.name, s2.name) < 0 ?
```

```
            comparison.theFirstComesFirst :
            comparison.theSecondComesFirst);
      }
      public override string ToString(  )
      {
         return name;
      }
      private string name;
   }

   public class Test
   {
      public static void Main(  )
      {
         // создать по паре объектов студентов и собак
         // и разместить их в объектах Pair
         Student Jesse = new Student("Jesse");
         Student Stacey = new Student ("Stacey");
         Dog Milo = new Dog(65);
         Dog Fred = new Dog(12);

         Pair studentPair = new Pair(Jesse,Stacey);
         Pair dogPair = new Pair(Milo, Fred);
         Console.WriteLine("studentPair\t\t\t: {0}",
            studentPair.ToString(  ));
         Console.WriteLine("dogPair\t\t\t\t: {0}",
            dogPair.ToString(  ));

         // создать объекты делегатов
         Pair.WhichIsFirst  theStudentDelegate =
            new Pair.WhichIsFirst(
            Student.WhichStudentComesFirst);

         Pair.WhichIsFirst theDogDelegate =
            new Pair.WhichIsFirst(
            Dog.WhichDogComesFirst);

         // сортировка с использованием делегатов
         studentPair.Sort(theStudentDelegate);
         Console.WriteLine("После сортировки studentPair\t\t: {0}",
            studentPair.ToString(  ));
         studentPair.ReverseSort(theStudentDelegate);
         Console.WriteLine("После обратной сортировки studentPair\t: {0}",
            studentPair.ToString(  ));

         dogPair.Sort(theDogDelegate);
         Console.WriteLine("После сортировки dogPair\t\t: {0}",
            dogPair.ToString(  ));
         dogPair.ReverseSort(theDogDelegate);
         Console.WriteLine("После обратной сортировки dogPair\t: {0}",
            dogPair.ToString(  ));
      }
   }
```

```
}
```

Вывод:

```
studentPair                          : Jesse, Stacey
dogPair                              : 65, 12
После сортировки studentPair         : Jesse, Stacey
После обратной сортировки studentPair  : Stacey, Jesse
После сортировки dogPair             : 12, 65
После обратной сортировки dogPair    : 65, 12
```

Программа Test создает два объекта класса Student и два объекта класса Dog, которые помещаются в объекты Pair. Конструктор Student() принимает строку с именем студента, а конструктор Dog() – вес собаки, выраженный целым числом.

```
Student Jesse = new Student("Jesse");
Student Stacey = new Student ("Stacey");
Dog Milo = new Dog(65);
Dog Fred = new Dog(12);

Pair studentPair = new Pair(Jesse,Stacey);
Pair dogPair = new Pair(Milo, Fred);
Console.WriteLine("studentPair\t\t\t: {0}", studentPair.ToString());
Console.WriteLine("dogPair\t\t\t\t: {0}", dogPair.ToString());
```

Затем выводится содержимое двух контейнеров Pair, демонстрирующее, в каком порядке расположены поступившие объекты:

```
studentPair          : Jesse, Stacey
dogPair              : 65, 12
```

Как и следовало ожидать, они хранятся в порядке поступления. Далее создаются объекты двух делегатов:

```
Pair.WhichIsFirst  theStudentDelegate =
   new Pair.WhichIsFirst(Student.WhichStudentComesFirst);

Pair.WhichIsFirst theDogDelegate =
   new Pair.WhichIsFirst(Dog.WhichDogComesFirst);
```

Первый делегат, theStudentDelegate, создается передачей соответствующего статического метода класса Student. Второй, theDogDelegate, передачей статического метода класса Dog.

Теперь делегаты являются объектами, которые могут быть переданы методам. Вначале они передаются методу Sort() объекта Pair, а затем – методу ReverseSort(). Результаты выводятся на экран:

```
После сортировки studentPair         : Jesse, Stacey
После обратной сортировки studentPair  : Stacey, Jesse
После сортировки dogPair             : 12, 65
После обратной сортировки dogPair    : 65, 12
```

Статические делегаты

У программы из примера 12.1 есть один недостаток. Вызывающему классу, в данном случае классу Test, приходится создавать экземпляры делегатов, нужных ему для упорядочивания объектов в объекте Pair. Было бы гораздо удобнее получать делегат прямо от класса Student или Dog. Этого нетрудно добиться, если включить в каждый класс собственный статический делегат. Внесем изменения в класс Student, добавив в него делегат:

```
public static readonly Pair.WhichIsFirst OrderStudents =
   new Pair.WhichIsFirst(Student.WhichStudentComesFirst);
```

Будет создан статический делегат с именем OrderStudents, доступный только для чтения.

Модификатор readonly в определении OrderStudents гарантирует, что после создания этого статического поля оно не будет изменено.

Аналогичным образом создается делегат в классе Dog:

```
public static readonly Pair.WhichIsFirst OrderDogs =
    new Pair.WhichIsFirst(Dog. WhichDogComesFirst);
```

Теперь делегаты – это статические поля соответствующих классов. Каждое из них жестко связано с соответствующим методом класса. Эти делегаты можно вызывать без создания локальных экземпляров. Достаточно будет передать методам класса Pair статический делегат соответствующего класса:

```
studentPair.Sort(Student.OrderStudents);
Console.WriteLine("После сортировки studentPair\t\t: {0}",
studentPair.ToString());
 studentPair.ReverseSort(Student.OrderStudents);
 Console.WriteLine("После обратной сортировки studentPair\t: {0}",
    studentPair.ToString());

 dogPair.Sort(Dog.OrderDogs);
 Console.WriteLine("После сортировки dogPair\t\t: {0}", dogPair.ToString());
 dogPair.ReverseSort(Dog.OrderDogs);
 Console.WriteLine("После обратной сортировки dogPair\t: {0}",
dogPair.ToString());
```

После внесения этих изменений в программу результат ее работы не изменится.

Делегаты как свойства

Статические делегаты имеют один недостаток: их экземпляры должны быть созданы независимо от того, будут ли они использованы, как

это было с классами Student и Dog в предыдущем примере. Положение можно исправить, если заменить статические поля делегатов на свойства.

Уберем из класса Student определение:

```
public static readonly Pair.WhichIsFirst OrderStudents =
   new Pair.WhichIsFirst(Student.WhichStudentComesFirst);
```

и поставим вместо него:

```
public static Pair.WhichIsFirst OrderStudents
{
    get
    {
        return new Pair.WhichIsFirst(WhichStudentComesFirst);
    }
}
```

Аналогично поступим со статическим полем класса Dog:

```
public static Pair.WhichIsFirst OrderDogs
{
    get
    {
        return new Pair.WhichIsFirst(WhichDogComesFirst);
    }
}
```

Передача делегатов в качестве параметров остается прежней:

```
studentPair.Sort(Student.OrderStudents);
dogPair.Sort(Dog.OrderDogs);
```

При обращении к свойству OrderStudent создается делегат:

```
return new Pair.WhichIsFirst(WhichStudentComesFirst);
```

Основным достоинством такого подхода является тот факт, что делегат не создается, пока не будет запрошен. Это позволяет классу Test самому решать, когда ему нужен делегат, причем ответственность за создание делегата по-прежнему лежит на классе Student (или Dog).

Определение порядка вызова методов с помощью массива делегатов

Делегаты помогают программисту создавать системы, в которых пользователь динамически определяет порядок выполнения действий. Предположим, имеется система обработки изображений, в которой изображение может быть подвергнуто ряду операций, таких как изме-

нение резкости, поворот, фильтрация и т. д. Предположим также, что порядок применения этих операций к изображению достаточно важен. Пользователь хочет выбрать эффекты из меню и сообщить программе, в каком порядке нужно их применить к изображению.

Чтобы удовлетворить сформулированным требованиям, можно для каждой операции создать делегат и разместить их в коллекции, например в массиве, в порядке, указанном пользователем. Когда все делегаты созданы и сохранены в коллекции, достаточно перебрать его элементы в цикле, поочередно вызывая каждый делегированный метод.

Начнем с создания класса `Image`, представляющего изображение, обрабатываемое классом `ImageProcessor`:

```
public class Image
{
    public Image()
    {
        Console.WriteLine("Изображение создано");
    }
}
```

Будем считать, что объект `Image` представляет файл в формате *.gif* или *.jpeg* (или в каком-нибудь другом).

В классе `ImageProcessor` объявим делегат. Программист может выбирать любые параметры и любой тип возвращаемого значения. Пусть в данном примере делегат инкапсулирует метод, который не принимает аргументов и возвращает `void`:

```
public delegate void DoEffect();
```

Теперь нужно объявить ряд методов, каждый из которых обрабатывает объект `Image` и соответствует сигнатуре и возвращаемому типу делегата:

```
public static void Blur()
 {
     Console.WriteLine("Растушевка изображения");
 }

 public static void Filter()
 {
     Console.WriteLine("Наложение фильтра");
 }
 public static void Sharpen()
 {
     Console.WriteLine("Настройка резкости");
 }

 public static void Rotate()
 {
     Console.WriteLine("Вращение изображения");
 }
```

В реальной ситуации эти методы должны выполнять фактические действия по растушевке, наложению фильтра, настройке резкости и повороту изображения соответственно. Конечно, они будут очень сложны.

Классу `ImageProcessor` нужен массив для хранения делегатов, выбранных пользователем, а также переменная, отслеживающая размер массива. Естественно, нужна и переменная для хранения изображения:

```
DoEffect[] arrayOfEffects;
Image image;
int numEffectsRegistered = 0;
```

Далее, классу `ImageProcessor` необходим метод для записи делегатов в массив:

```
public void AddToEffects(DoEffect theEffect)
{
    if (numEffectsRegistered >= 10)
    {
        throw new Exception("Слишком много элементов в массиве");
    }
    arrayOfEffects[numEffectsRegistered++] = theEffect;

}
```

Еще ему нужен метод, вызывающий делегированные методы по очереди:

```
public void ProcessImages()
{
    for (int i = 0;i < numEffectsRegistered;i++)
    {
        arrayOfEffects[i]();
    }
}
```

Наконец, остается объявить статические делегаты, которые будет вызывать пользователь, и привязать к ним методы обработки изображения:

```
public DoEffect BlurEffect = new DoEffect(Blur);
public DoEffect SharpenEffect = new DoEffect(Sharpen);
public DoEffect FilterEffect = new DoEffect(Filter);
public DoEffect RotateEffect = new DoEffect(Rotate);
```

В реальном приложении, где таких эффектов будут десятки, разумнее реализовать их в виде свойств, а не статических методов. Тогда можно будет сэкономить на том, что свойства создаются лишь по мере необходимости. Правда, эта экономия достигается за счет некоторого усложнения программы.

Программа пользователя, как правило, включает в себя компонент, реализующий пользовательский интерфейс. В примере 12.2 имитируется выбор эффектов, после чего программа добавляет их в массив и вызывает метод `ProcessImage()`.

Пример 12.2. Массив делегатов

```
namespace Programming_CSharp
{
   using System;

   // изображение, подвергающееся обработке
   public class Image
   {
      public Image()
      {
         Console.WriteLine("Изображение создано");
      }
   }

   public class ImageProcessor
   {
      // объявить делегат
      public delegate void DoEffect();

      // создать различные статические делегаты,
      // связанные с методами класса
      public DoEffect BlurEffect =
         new DoEffect(Blur);
      public DoEffect SharpenEffect =
         new DoEffect(Sharpen);
      public DoEffect FilterEffect =
         new DoEffect(Filter);
      public DoEffect RotateEffect =
         new DoEffect(Rotate);

      // конструктор инициализирует изображение и массив
      public ImageProcessor(Image image)
      {
         this.image = image;
         arrayOfEffects = new DoEffect[10];
      }

      // в реальном приложении был бы использован
      // более гибкий класс коллекции
      public void AddToEffects(DoEffect theEffect)
      {
         if (numEffectsRegistered >= 10)
         {
            throw new Exception(
               "Слишком много элементов в массиве");
         }
         arrayOfEffects[numEffectsRegistered++]
            = theEffect;
```

```
    }

    // методы обработки изображения...
    public static void Blur()
    {
       Console.WriteLine("Растушевка изображения");
    }

    public static void Filter()
    {
       Console.WriteLine("Наложение фильтра");
    }

    public static void Sharpen()
    {
       Console.WriteLine("Регулировка контрастности");
    }

    public static void Rotate()
    {
       Console.WriteLine("Вращение изображения");
    }

    public void ProcessImages()
    {
       for (int i = 0;i < numEffectsRegistered;i++)
       {
          arrayOfEffects[i]();
       }
    }

    // закрытые переменные класса...
    private DoEffect[] arrayOfEffects;

    private Image image;
    private int numEffectsRegistered = 0;

}

// тестовый класс
public class Test
{
    public static void Main()
    {
       Image theImage = new Image();

       // ради простоты пользовательский интерфейс отсутствует,
       // программа просто добавляет методы в массив и
       // вызывает класс-обработчик, вызывающий
       // методы в порядке добавления
       ImageProcessor theProc =
          new ImageProcessor(theImage);
       theProc.AddToEffects(theProc.BlurEffect);
       theProc.AddToEffects(theProc.FilterEffect);
       theProc.AddToEffects(theProc.RotateEffect);
```

```
            theProc.AddToEffects(theProc.SharpenEffect);
            theProc.ProcessImages();
        }
    }
}
```

Вывод:

```
Изображение создано
Растушевка изображения
Наложение фильтра
Вращение изображения
Регулировка контрастности
```

В классе `Test` из примера 12.2 создается экземпляр класса `ImageProcessor`, и в массив добавляются различные эффекты. Если пользователь предпочитает сначала произвести растушевку изображения, а потом применить фильтр, то именно в таком порядке делегаты добавляются в массив. Аналогичным образом, если пользователь захочет повторить какую-нибудь операцию, в коллекцию можно еще раз добавить соответствующий делегат.

В реальном приложении операции могут быть представлены в виде списка, позволяющего пользователю менять порядок следования элементов. Если он переместит какие-либо эффекты в списке, программе будет достаточно переместить элементы коллекции. Более того, список операций можно хранить в базе данных, загружая их динамически и создавая экземпляры делегатов в соответствии с записями в базе данных.

Делегаты обеспечивают гибкость реализации, позволяя динамически определять, какие методы следует вызвать, в каком порядке и как часто.

Множественное делегирование

Иногда бывает желательно выполнять *множественное делегирование* (*multicast*), то есть вызов двух методов из одного делегата. Это становится особенно актуально при обработке событий (обсуждаемых далее в этой главе).

Поставленная цель – иметь делегат, вызывающий несколько методов, – отличается от ситуации, в которой присутствует коллекция делегатов, каждый из которых вызывает один метод. В предыдущем примере коллекция использовалась для упорядочивания различных делегатов. Допускалось многократное добавление одного и того же делегата в коллекцию, а также переупорядочивание делегатов в коллекции с целью изменения порядка их вызовов.

При множественном делегировании создается один делегат, вызывающий несколько инкапсулированных методов. Например, когда пользователь щелкает по кнопке, может потребоваться, чтобы приложение

предприняло несколько действий. Конечно, можно реализовать это требование с помощью коллекции делегатов, однако проще и элегантнее организовать множественное делегирование.

Множественное делегирование можно использовать с делегатами, возвращающими значение (имеется в виду значение, отличное от void). В этом случае вы получите только одно значение: значение последнего вызванного делегата.

Два делегата можно объединить с помощью операции сложения (+). Результатом будет новый множественный делегат, вызывающий методы обоих объединенных делегатов. Например, пусть Writer и Logger являются делегатами. В следующей строчке кода создается новый множественный делегат по имени myMulticastDelegate:

```
myMulticastDelegate = Writer + Logger;
```

К множественному делегату можно добавить делегат с помощью оператора +=. Делегат, стоящий справа от знака операции, добавляется к множественному делегату, стоящему слева. Например, если Transmitter и myMulticastDelegate являются делегатами, то следующая строчка кода соединит их в один:

```
myMulticastDelegate += Transmitter;
```

Чтобы понять, как создаются и используются множественные делегаты, разберем пример. В примере 12.3 создается класс MyClassWithDelegate, который объявляет делегат, принимающий строку и возвращающий void:

```
public delegate void StringDelegate(string s);
```

Затем определяется класс MyImplementingClass с тремя методами, каждый из которых принимает строку и возвращает void: WriteString, LogString и TransmitString. Первый выводит строку в стандартный поток вывода, второй имитирует запись в регистрационный журнал, а третий имитирует передачу строки в Интернете. Для вызова соответствующих методов создаются объекты делегатов:

```
Writer("Строка передана методу Writer\n");
Logger("Строка передана методу Logger\n");
Transmitter("Строка передана методу Transmitter\n");
```

Чтобы проследить за объединением делегатов, создадим еще один объект делегата:

```
MyClassWithDelegate.StringDelegate myMulticastDelegate;
```

и присвоим ему результат «сложения» двух имеющихся:

```
myMulticastDelegate = Writer + Logger;
```

С помощью операции += добавим к нему еще один делегат:

```
myMulticastDelegate += Transmitter;
```

Наконец, из делегата удаляется один, для чего используется операция -=:

```
myMulticastDelegate -= Logger;
```

Пример 12.3. Объединение делегатов

```
namespace Programming_CSharp
{
   using System;

   public class MyClassWithDelegate
   {
      // объявление делегата
      public delegate void StringDelegate(string s);

   }

   public class MyImplementingClass
   {
      public static void WriteString(string s)
      {
         Console.WriteLine("Вывод строки {0}", s);
      }
      public static void LogString(string s)
      {
         Console.WriteLine("Запись строки в журнал {0}", s);
      }

      public static void TransmitString(string s)
      {
         Console.WriteLine("Передача строки {0}", s);
      }

   }

   public class Test
   {
      public static void Main()
      {
         // определить три делегата StringDelegate
         MyClassWithDelegate.StringDelegate
            Writer, Logger, Transmitter;

         // определить еще один объект StringDelegate,
         // который будет множественным делегатом
         MyClassWithDelegate.StringDelegate
            myMulticastDelegate;

         // создать экземпляры первых трех делегатов,
         // передавая методы для инкапсуляции
```

```
    Writer = new MyClassWithDelegate.StringDelegate(
       MyImplementingClass.WriteString);
    Logger = new MyClassWithDelegate.StringDelegate(
       MyImplementingClass.LogString);
    Transmitter =
       new MyClassWithDelegate.StringDelegate(
       MyImplementingClass.TransmitString);

    // вызвать делегированный метод Writer
    Writer("Строка передана методу Writer\n");

    // вызвать делегированный метод Logger
    Logger("Строка передана методу Logger\n");

    // вызвать делегированный метод Transmitter
    Transmitter("Строка передана методу Transmitter\n");

    // сообщить о готовности объединить
    // два делегата в один множественный
    Console.WriteLine(
       "myMulticastDelegate = Writer + Logger");

    // объединить два делегата и
    // присвоить результат делегату myMulticastDelegate
    myMulticastDelegate = Writer + Logger;

    // вызвать делегированные методы -
    // будут вызваны два метода
    myMulticastDelegate(
       "Первая строка передана методу Collector");

    // сообщить о готовности добавить
    // третий делегат к множественному делегату
    Console.WriteLine(
       "\nmyMulticastDelegate += Transmitter");

    // добавить третий делегат
    myMulticastDelegate += Transmitter;

    // вызвать три делегированных метода
    myMulticastDelegate(
       "Вторая строка передана методу Collector");

    // сообщить о готовности удалить
    // один делегат
    Console.WriteLine(
       "\nmyMulticastDelegate -= Logger");

    // удалить один делегат
    myMulticastDelegate -= Logger;

    // вызвать два оставшихся
    // делегированных метода
    myMulticastDelegate(
       "Третья строка передана методу Collector");
}
```

```
    }
}
```

Вывод:

```
Вывод строки Строка передана методу Writer

Запись строки в журнал Строка передана методу Logger

Передача строки Строка передана методу Transmitter

myMulticastDelegate = Writer + Logger
вывод строки Первая строка передана методу Collector
Запись строки в журнал Первая строка передана методу Collector

myMulticastDelegate += Transmitter
Вывод строки Вторая строка передана методу Collector
Запись строки в журнал Вторая строка передана методу Collector
Передача строки Вторая строка передана методу Collector

myMulticastDelegate -= Logger
Вывод строки Третья строка передана методу Collector
Передача строки Третья строка передана методу Collector
```

В тестирующей части программы из примера 12.3 определяются объекты делегатов и вызываются первые три из них (`Writer`, `Logger` и `Transmitter`). Четвертому делегату, `myMulticastDelegate`, присваивается объединение первых двух, после чего он тоже вызывается. Это приводит к вызову обоих делегированных методов. Третий делегат добавляется к делегату `myMulticastDelegate`, и, когда тот вызывается, срабатывают все три метода. Наконец, делегат `Logger` удаляется из множественного делегата. Теперь вызов делегата `myMulticastDelegate` заканчивается вызовом двух оставшихся методов.

Преимущества множественного делегирования легче оценить, когда речь заходит о событиях, рассматриваемых в следующем разделе. Когда возникает некоторое событие, например нажатие кнопки, связанный с ним делегат может вызвать целый ряд обработчиков для реакции на данное событие.

События

Графические пользовательские интерфейсы (GUI), система Windows и веб-броузеры (например, выпущенные компанией Microsoft) требуют, чтобы программы реагировали на *события* (*event*). Событием может быть нажатие кнопки, выбор команды меню, завершение операции передачи файла и т. д. Короче говоря, если происходит что-то важное, программа должна отреагировать. Порядок возникновения событий предсказать невозможно. Система ждет возникновения события, а затем предпринимает действия по его обработке.

В среде GUI любые элементы управления могут *вызвать* событие. Например, если щелкнуть по кнопке, вызывается событие `Click`, а ес-

ли добавить элемент в раскрывающийся список, может быть вызвано событие `ListChanged`.

Реакция на событие является предметом заботы других классов, и она совсем не интересует класс, вызвавший событие. Кнопка как бы говорит: «По мне щелкнули», а реагировать должны другие классы.

Публикация и подписка

В языке C# любой объект может *опубликовать* набор событий, а другие классы могут на них *подписаться*. Когда публикующий класс вызывает событие, все подписавшиеся классы уведомляются об этом.

Подобное проектное решение является реализацией паттерна «издатель/подписчик», описанного в основополагающем труде «Паттерны проектирования» Гаммы и др. («Design Patterns», by Gamma, et al.), вышедшем в издательстве Addison Wesley в 1995 году.[1] Гамма формулирует предназначение данного образца следующим образом: «Определить между объектами зависимость «один-ко-многим», чтобы при смене состояния одного объекта все зависимые объекты уведомлялись об этом и автоматически меняли свои состояния».

Когда имеется такой механизм, объект говорит: «Вот события, о которых я могу уведомить». Другие классы подписываются, говоря: «Да, уведомите меня, когда оно произойдет». Например, кнопка уведомляет заинтересованных наблюдателей, когда пользователь щелкнет по ней. Кнопка называется *издателем*, потому что публикует событие `Click`, а другие классы называются *подписчиками*, потому что подписываются на это событие.

События и делегаты

События в C# обрабатываются с помощью делегатов. Класс-издатель определяет делегат, который должен быть реализован классами-подписчиками. Когда возникает событие, методы подписчиков вызываются через этот делегат.

Метод, реагирующий на события, называется *обработчиком события* (*event handler*). Программист объявляет обработчик события как любой другой делегат.

Существует соглашение, по которому обработчики событий в .NET Framework возвращают `void` и принимают два параметра. Первый –

[1] Гамма, Хелм, Джонсон, Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб:Питер, 2000 г.

это *источник* события, то есть объект-издатель. Вторым параметром передается объект, производный от класса EventArgs. Рекомендуется, чтобы обработчики событий, создаваемые всеми программистами, следовали этому соглашению.

EventArgs является классом, базовым для всех классов событий. В отличие от своего конструктора, этот класс наследует все методы от класса Object, хотя он и добавляет открытое статическое поле empty, представляющее событие без состояния (чтобы обеспечить эффективную обработку таких событий). Классы, производные от EventArgs, содержат информацию о возникшем событии.

Событие является свойством класса, опубликовавшего его. Ключевое слово event управляет доступом классов-подписчиков к событию. Оно предназначено для обеспечения идиомы «публикация/подписка».

Предположим, создается класс Clock, который с помощью события уведомляет классы-подписчики о том, что местное время увеличилось на одну секунду. Назовем это событие OnSecondChange. Событие и тип делегата для обработки события определяются следующим образом:

```
[атрибуты] [модификаторы] event тип
        имя
```

Например:

```
public event SecondChangeHandler OnSecondChange;
```

Здесь не указаны атрибуты (которые обсуждаются в главе 18). В качестве модификатора указывается ключевое слово abstract, new, override, static, virtual или один из четырех модификаторов права доступа, в данном случае public.

За модификатором стоит ключевое слово event.

Тип – это делегат, который должен быть связан с событием, в данном случае SecondChangeHandler.

Имя – имя события, в рассматриваемом примере это OnSecondChange. Принято начинать имена событий с префикса On.

Итак, приведенное объявление утверждает, что OnSecondChange является событием, реализованным с помощью делегата SecondChangeHandler.

Объявление этого делегата выглядит так:

```
public delegate void SecondChangeHandler(
    object clock,
    TimeInfoEventArgs timeInformation
    );
```

Здесь объявляется делегат. Как говорилось выше, в соответствии с соглашением обработчик события должен возвращать void и принимать два параметра: источник события (объект Clock) и объект класса, про-

изводного от EventArgs (в данном примере TimeInfoEventArgs). Класс TimeInfoEventArgs определяется следующим образом:

```
public class TimeInfoEventArgs : EventArgs
{
     public TimeInfoEventArgs(int hour, int minute, int second)
     {
         this.hour = hour;
         this.minute = minute;
         this.second = second;
     }

     public readonly int hour;
     public readonly int minute;
     public readonly int second;
}
```

Объект TimeInfoEventArgs будет содержать информацию о текущем времени (часы, минуты и секунды). Он определяет конструктор и три открытые переменные, доступные только для чтения.

Кроме делегата и события класс Clock содержит три переменные — hour, minute и second — и один метод Run():

```
public void Run()
{
    for(;;)
    {
        // ждать 10 миллисекунд
        Thread.Sleep(10);

        // получить текущее время
        System.DateTime dt = System.DateTime.Now;

        // если количество секунд изменилось,
        // уведомить подписчиков
        if (dt.Second != second)
        {
            // создать объект TimeInfoEventArgs
            // для передачи подписчику
            TimeInfoEventArgs timeInformation =
                new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second);

            // если есть подписчики, уведомить их
            if (OnSecondChange != null)
            {
                OnSecondChange(this,timeInformation);
            }
        }

        // обновить состояние
        this.second = dt.Second;
        this.minute = dt.Minute;
```

```
            this.hour = dt.Hour;
        }
    }
```

В методе Run() работает бесконечный цикл for, в котором периодически проверяется системное время. Если оно отличается от значения, хранящегося в объекте Clock, он уведомляет подписчиков и обновляет свое состояние.

Первое, что делает метод, – переходит в режим ожидания на 10 миллисекунд:

```
Thread.Sleep(10);
```

Здесь вызывается статический метод класса Thread, принадлежащего пространству имен System.Threading. Более подробно этот класс обсуждается в главе 20. Вызов метода Sleep() не дает циклу for занимать практически все время процессора.

Прождав 10 миллисекунд, метод проверяет текущее время:

```
System.DateTime dt = System.DateTime.Now;
```

Приблизительно один раз за сто проверок увеличивается значение переменной second. Когда метод обнаруживает это, он оповещает своих подписчиков. С этой целью создается новый объект TimeInfoEventArgs:

```
if (dt.Second != second)
{
   // создать объект TimeInfoEventArgs
   // для передачи подписчику
   TimeInfoEventArgs timeInformation =
      new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second);
```

Затем метод Run() уведомляет подписчиков, вызывая событие OnSecondChange:

```
   // если есть подписчики, уведомить их
   if (OnSecondChange != null)
   {
      OnSecondChange(this,timeInformation);
   }
}
```

Если у события нет зарегистрированных подписчиков, оно имеет значение null. Условие в операторе if сравнивает событие со значением null. Следует убедиться в наличии подписчиков прежде, чем вызывать событие OnSecondChange.

Читатель помнит, что обработчик события OnSecondChange принимает два аргумента – источник события и объект класса, производного от EventArgs. Во фрагменте программы, приведенном выше, видно, что

первым аргументом является ссылка this, поскольку сам объект Clock и является источником события. Второй аргумент – это объект TimeInfoEventArgs, созданный строчкой раньше.

Вызов события приведет к вызову методов, зарегистрированных классом Clock с помощью делегата. Этот вопрос будет обсужден чуть позже.

После вызова события класс Clock обновляет свое состояние:

```
this.second = dt.Second;
this.minute = dt.Minute;
this.hour = dt.Hour;
```

Обратите внимание, что этот код не обеспечивает безопасность при работе с несколькими потоками. Безопасность и синхронизация обсуждаются в главе 20.

Итак, остается лишь создать классы, подписывающиеся на это событие. Создадим два. Первый назовем DisplayClock. Его задача сводится к показу текущего времени на экране компьютера.

В рассматриваемом примере этот класс будет максимально упрощен и ограничится двумя методами. Первый, вспомогательный, называется Subscribe. Он будет подписывать класс на событие OnSecondChange класса Clock. Вторым методом будет обработчик события по имени TimeHasChanged:

```
public class DisplayClock
{
    public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(TimeHasChanged);
    }

    public void TimeHasChanged(
        object theClock, TimeInfoEventArgs ti)
    {
            Console.WriteLine("Текущее время: {0}:{1}:{2}",
                ti.hour.ToString(),
                ti.minute.ToString(),
                ti.second.ToString());
   }
}
```

Когда вызывается метод Subscribe(), он создает новый делегат SecondChangeHandler, передавая ему обработчик события TimeHasChanged(). Затем он подписывает этот делегат на событие OnSecondChange класса Clock.

Второй класс, реагирующий на событие, будет называться LogCurrentTime. В реальном приложении он регистрировал бы событие в фай-

ле журнала, но здесь, в демонстрационном примере, он будет выводить сообщение на экран:

```
public class LogCurrentTime
{
   public void Subscribe(Clock theClock)
    {
        theClock.OnSecondChange += new
Clock.SecondChangeHandler(WriteLogEntry);
    }

    // этот метод должен выводить информацию в файл, но здесь
    // он выводит ее на экран в демонстрационных целях;
    // этот объект не поддерживает никакое состояние
    public void WriteLogEntry(
        object theClock, TimeInfoEventArgs ti)
    {
        Console.WriteLine("Записано в журнал: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
    }
}
```

Хотя в данном примере классы-подписчики имеют много общего, в реальном приложении на событие может подписаться любое количество совершенно непохожих классов.

Обратим внимание, что события добавляются к объекту операцией `+=`. Такой подход позволяет впоследствии добавить новые события к событию `OnSecondChange` объекта `Clock`, не разрушая при этом уже существующие. Когда класс `LogCurrentTime` подписывается на событие `OnSecondChange`, нельзя допустить, чтобы оно «забыло», что на него уже подписан класс `DisplayClock`.

Итак, нужно создать класс `Clock`, а также класс `DisplayClock` и заставить их подписаться на событие. Затем создадим класс `LogCurrentTime` и тоже подпишем его на событие. Наконец, выполним класс `Clock`. Все это демонстрируется в примере 12.4.

Пример 12.4. Работа с событиями

```
namespace Programming_CSharp
{
   using System;
   using System.Threading;

   // класс для хранения информации о событии;
   // в данном случае он содержит только сведения,
   // полученные от класса Clock, но мог бы
   // хранить и состояние
   public class TimeInfoEventArgs : EventArgs
   {
```

```
   public TimeInfoEventArgs(int hour, int minute, int second)
   {
      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }
   public readonly int hour;
   public readonly int minute;
   public readonly int second;
}

// главное действующее лицо - класс, за которым наблюдают
// другие классы; он публикует одно событие, OnSecondChange;
// наблюдатели подписываются на это событие
public class Clock
{
   // делегат, который должен быть реализован подписчиками
   public delegate void SecondChangeHandler
      (
      object clock,
      TimeInfoEventArgs timeInformation
      );

   // публикуемое событие
   public event SecondChangeHandler OnSecondChange;

   // запустить класс Clock;
   // он будет вызывать событие каждую секунду
   public void Run()
   {
      for(;;)
      {
         // ждать 10 миллисекунд
         Thread.Sleep(10);

         // получить текущее время
         System.DateTime dt = System.DateTime.Now;

         // если количество секунд изменилось,
         // уведомить подписчиков
         if (dt.Second != second)
         {
            // создать объект TimeInfoEventArgs
            // для передачи подписчику
            TimeInfoEventArgs timeInformation =
               new TimeInfoEventArgs(
                  dt.Hour,dt.Minute,dt.Second);

            // если есть подписчики, уведомить их
            if (OnSecondChange != null)
            {
               OnSecondChange(
                  this,timeInformation);
            }
```

```
            }
            // обновить состояние
            this.second = dt.Second;
            this.minute = dt.Minute;
            this.hour = dt.Hour;

         }
      }
      private int hour;
      private int minute;
      private int second;
   }

   // наблюдатель; класс DisplayClock подписывается на
   // событие класса Clock; его задача -
   // выводить на консоль текущее время
   public class DisplayClock
   {
      // подписаться на событие SecondChangeHandler
      // класса Clock
      public void Subscribe(Clock theClock)
      {
         theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(TimeHasChanged);
      }

      // метод, реализующий делегируемую функциональность
      public void TimeHasChanged(
         object theClock, TimeInfoEventArgs ti)
      {
         Console.WriteLine("Текущее время: {0}:{1}:{2}",
            ti.hour.ToString(),
            ti.minute.ToString(),
            ti.second.ToString());
      }
   }

   // второй подписчик, который должен выводить информацию в файл
   public class LogCurrentTime
   {
      public void Subscribe(Clock theClock)
      {
         theClock.OnSecondChange +=
            new Clock.SecondChangeHandler(WriteLogEntry);
      }

      // этот метод должен выводить информацию в файл, но здесь
      // он выводит ее на экран в демонстрационных целях;
      // этот объект не поддерживает никакое состояние
      public void WriteLogEntry(
         object theClock, TimeInfoEventArgs ti)
      {
```

```
            Console.WriteLine("Записано в журнал: {0}:{1}:{2}",
               ti.hour.ToString(),
               ti.minute.ToString(),
               ti.second.ToString());
         }
      }

      public class Test
      {
         public static void Main()
         {
            // создать новый объект Clock
            Clock theClock = new Clock();

            // создать объект DisplayClock и заставить его
            // подписаться на событие только что созданного класса
            DisplayClock dc = new DisplayClock();
            dc.Subscribe(theClock);

            // создать объект LogCurrentTime и заставить его
            // подписаться на событие класса Clock
            LogCurrentTime lct = new LogCurrentTime();
            lct.Subscribe(theClock);

            // запустить класс Clock
            theClock.Run();
         }
      }
   }
```

Вывод:

```
Текущее время: 14:53:56
Записано в журнал: 14:53:56
Текущее время: 14:53:57
Записано в журнал: 14:53:57
Текущее время: 14:53:58
Записано в журнал: 14:53:58
Текущее время: 14:53:59
Записано в журнал: 14:53:59
Текущее время: 14:54:0
Записано в журнал: 14:54:0
```

«Чистым» результатом этой программы является создание двух классов, `DisplayClock` и `LogCurrentTime`, каждый из которых подписан на событие третьего класса, а именно `Clock.OnSecondChange`.

Отделение издателей от подписчиков

Класс `Clock` и сам мог бы выводить время, не вызывая никакое событие, так к чему все эти хлопоты с делегатами? Достоинство идиомы «публикация/подписка» состоит в возможности уведомить любое количество классов о возникшем событии. Классам-подписчикам нет дела

до того, как работает класс `Clock`, а ему – до того, как они отреагируют на событие. Аналогичным образом кнопка может опубликовать событие `Onclick`, и любое количество самых разных объектов может подписаться на него и получать уведомление каждый раз, когда пользователь щелкнет по кнопке.

Издатель и подписчик отделены друг от друга делегатом. Это очень удобно, поскольку такое решение позволяет писать гибкий и устойчивый код. Класс `Clock` может изменить способ определения времени, не нарушая при этом работу классов-подписчиков. Со своей стороны, подписчики могут менять реакцию на событие, не нарушая работу класса `Clock`. Независимое существование классов облегчает сопровождение кода.

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-038-3, название «Программирование на C#, 2-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

II

Программирование на C#

13

Создание Windows-приложений

В предыдущих главах для демонстрации функциональных возможностей C# и среды Common Language Runtime использовались консольные приложения. Написать консольное приложение очень просто, но теперь настало время вспомнить о главной причине, побуждающей программистов изучать язык C#. Это необходимость создавать приложения для Windows и веб-приложения.

На заре возникновения Windows каждое приложение работало на персональном компьютере в полной изоляции. Со временем разработчики открыли для себя преимущества распределения приложений по сети, когда пользовательский интерфейс реализован на одном компьютере, а база данных – на другом. Такое разделение ответственности обычно называется двухуровневым подходом к разработке приложения или подходом «клиент-сервер». Впоследствии разработчики стали пользоваться веб-серверами для хранения объектов, обрабатывающих обращения клиентов к базам данных. Так возникли трехуровневые и вообще n-уровневые подходы.

Когда Всемирная паутина только зарождалась, имело место четкое разделение программ на Windows-приложения и веб-приложения. Первые работали на персональных компьютерах или в локальных сетях, в то время как вторые работали на удаленных серверах, а обращаться к ним нужно было с помощью броузеров. Теперь граница между этими видами приложений достаточно размыта, поскольку Windows-приложения могут пользоваться услугами Сети. Многие недавно созданные приложения включают в себя логическую часть, выполняемую на компьютере клиента, сервер баз данных и компоненты сторонних производителей, выполняемые на удаленных компьютерах в Сети. Традиционные «настольные» приложения, такие как Excel и Outlook, в настоящее

время способны естественным образом интегрировать данные, полученные из Сети, а веб-приложения могут распределять некоторые виды обработки данных между пользовательскими компонентами.

Быть может, последним важным различием между Windows-приложениями и веб-приложениями является способ взаимодействия с пользователем, то есть распределение ответственности за реализацию пользовательского интерфейса. Реализует ли приложение свой пользовательский интерфейс при помощи броузера, или же это интерфейс является неотъемлемой частью приложения, исполняемого на локальном компьютере?

У веб-приложения масса достоинств. Самое очевидное – к нему может обратиться любой броузер, сумевший соединиться с сервером. Кроме того, все изменения в него вносятся на сервере, и нет необходимости рассылать клиентам новые версии библиотек динамической компоновки (DLL).

С другой стороны, если приложение не предназначено для публикации в Сети, его разработчик может создать более удачный интерфейс или добиться более высокой производительности, если реализует его как самостоятельное приложение.

Платформа .NET предлагает два тесно связанных, но все-таки разных комплекта инструментальных средств для построения Windows-приложений и веб-приложений. Оба комплекта основываются на применении форм, поскольку многие приложения обладают интерфейсами, использующими формы и такие элементы управления, как кнопки, окна списков, текстовые поля и т. д.

Комплект для разработки веб-приложений называется Web Forms (он обсуждается в главе 15). Средства, позволяющие создавать Windows-приложения, называются Windows Forms, и они являются темой данной главы.

Автор готов предсказать, что различие между Web Forms и Windows Forms скоро сойдет на нет. Они имеют так много общего, что будет удивительно, если в следующей версии .NET они не сольются, образуя одну унифицированную среду разработки.

В следующих разделах будет продемонстрировано, как создать простую Windows-форму с помощью текстового редактора, например Notepad (Блокнота), или среды Visual Studio .NET. Затем читатель узнает, как строить более сложные Windows-приложения с помощью Visual Studio .NET, инструментов Windows Forms и приемов программирования на языке C#, описанных в предыдущих главах. Заканчивается данная глава кратким введением в Documentation Comments (Документирующие комментарии) – новую технологию, основанную на XML и предназначенную для документирования приложений, а также введением в развертывание приложений .NET.

Создание простой формы Windows

Windows Forms – это средство для построения приложений Windows. Платформа .NET Framework предоставляет широкую поддержку разработки приложений такого типа, и Windows Forms является важнейшим элементом этой поддержки. Как и следовало ожидать, в среде Windows Forms используется понятие формы. Идея, заимствованная из популярнейшей среды Visual Basic (VB), поддерживает технологию быстрой разработки приложений (RAD, Rapid Application Development). Возможно, C# является первой программной технологией, соединившей в себе RAD из среды VB и объектно-ориентированные функциональные возможности языков семейства C.

Использование Блокнота

Среда Visual Studio .NET предлагает широкий набор инструментов, поддерживающих технологию drag-and-drop и позволяющих взаимодействовать со средой Windows Forms. Все же есть способ построить Windows-приложение, не обращаясь за помощью к интегрированой среде разработки Visual Studio .NET, но он довольно трудоемкий и занимает много времени.

Исключительно для демонстрации этого способа воспользуемся текстовым редактором Блокнот (Notepad) и создадим простое приложение на основе Windows Forms, которое выводит текст в окне, содержащем кнопку Cancel (Отмена). Результат работы этого приложения изображен на рис. 13.1.

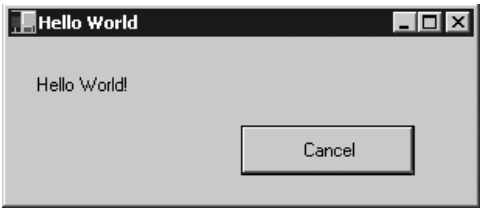

Рис. 13.1. *Перемещаемая Windows-форма*

Начнем с оператора `using`, определяющего пространство имен Windows Forms:

```
using System.Windows.Forms;
```

Суть создания приложения на основе Windows Forms заключается в определении формы, производной от `System.Windows.Forms.Form`:

```
public class HandDrawnClass : Form
```

Объект `Form` представляет любое окно, выведенное приложением на экран. Класс `Form` можно применять для создания стандартных окон, а

также всплывающих, инструментальных, диалоговых и прочих. По-видимому, в фирме Microsoft решили назвать этот класс словом «form» (форма), а не «window» (окно), чтобы подчеркнуть, что большинство окон имеют интерактивный компонент, содержащий элементы управления, позволяющие взаимодействовать с пользователями.

Все необходимые элементы управления (статический текст, кнопки, списки и т. д.) находятся в пространстве имен `Windows.Forms`. В среде Visual Studio .NET программист имеет возможность перетаскивать эти объекты мышью в окно проекта, а здесь их приходится объявлять в программе.

Для начала объявим два элемента управления, статический текст (label) «Hello World» и кнопку, которая позволит завершить работу приложения:

```
private System.Windows.Forms.Label lblOutput;
private System.Windows.Forms.Button btnCancel;
```

Теперь с помощью конструктора объекта `Form` можно создать объекты этих элементов управления:

```
this.lblOutput = new System.Windows.Forms.Label();
this.btnCancel = new System.Windows.Forms.Button();
```

Далее присвоим текст «Hello World» заголовку формы:

```
this.Text = "Hello World";
```

Необходимо заметить, что приведенные выше фрагменты программы находятся внутри конструктора `HandDrawnClass`, поэтому ключевое слово `this` относится к самой форме.

Установим местоположение и размер элемента управления, а также текст, который она будет содержать:

```
lblOutput.Location = new System.Drawing.Point (16, 24);
lblOutput.Text = "Hello World!";
lblOutput.Size = new System.Drawing.Size (216, 24);
```

Местоположение представлено объектом `System.Drawing.Point`, чей конструктор принимает в качестве аргументов горизонтальную и вертикальную координаты. Размер устанавливается с помощью объекта `Size`. Его конструктор принимает два целых числа, соответствующих ширине и высоте элемента управления.

Платформа .NET Framework предоставляет программисту пространство имен `System.Drawing`, которое инкапсулирует графические функции Win32 GDI+. Большая часть библиотеки классов платформы .NET состоит из классов, инкапсулирующих методы Win32 как объекты.

Аналогичным образом установим местоположение, размер и текст объекта Button:

```
btnCancel.Location = new System.Drawing.Point (150,200);
btnCancel.Size = new System.Drawing.Size (112, 32);
btnCancel.Text = "&Cancel";
```

Кроме того, для кнопки следует создать обработчик события. Как было сказано в главе 12, события (в данном случае щелчок кнопки) реализуются с помощью делегатов. Класс-издатель (Button) определяет делегат (System.EventHandler), который должен быть реализован классом-подписчиком (формой).

Делегированный метод может быть назван как угодно, но он должен иметь тип возвращаемого значения void и два формальных параметра: объект sender и объект типа SystemEventArgs, обычно называемый именем e:

```
protected void btnCancel_Click (object sender, System.EventArgs e)
{
   //...
}
```

Обработчик события регистрируется за два шага. Во-первых, создается новый делегат System.EventHandler, которому в качестве аргумента передается имя метода:

```
new System.EventHandler (this.btnCancel_Click);
```

На втором шаге с помощью операции += в список обработчиков события «щелчок кнопки» добавляется созданный объект делегата.

В следующей строке эти два шага объединены в один оператор:

```
btnCancel.Click +=
   new System.EventHandler (this.btnCancel_Click);
```

Теперь настало время определить размеры самой формы. Ее свойство AutoScaleBaseSize устанавливает базовый размер, который на этапе выполнения используется для вычисления коэффициента масштабирования формы. Свойство ClientSize устанавливает размер рабочей области формы, то есть размер формы, исключая его границы и строку заголовка (в интегрированной среде разработки программист выбирает эти значения в интерактивном режиме):

```
this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
this.ClientSize = new System.Drawing.Size (300, 300);
```

Наконец, не забудем добавить в форму созданные ранее элементы управления:

```
this.Controls.Add (this.btnCancel);
this.Controls.Add (this.lblOutput);
```

Зарегистрировав обработчик события, можно заняться его реализацией. В рассматриваемом примере щелчок кнопки Cancel завершит работу приложения с помощью статического метода Exit() класса Application:

```
protected void btnCancel_Click (object sender, System.EventArgs e)
{
   Application.Exit ();
}
```

Вот и все. Осталось снабдить программу точкой входа, чтобы можно было вызвать конструктор формы:

```
public static void Main()
{
    Application.Run(new HandDrawnClass());
}
```

Полностью исходный текст этой программы приведен в примере 13.1. Если запустить приложение, на экране откроется окно, в котором будет выведен текст. Нажатие кнопки Cancel закроет приложение.

Пример 13.1. Создание Windows-формы «вручную»

```
using System;
using System.Windows.Forms;

namespace ProgCSharp
{
   public class HandDrawnClass : Form
   {
      // объект статического текста Hello World
      private System.Windows.Forms.Label
         lblOutput;

      // кнопка Cancel
      private System.Windows.Forms.Button
         btnCancel;

      public HandDrawnClass()
      {
         // создать объекты
         this.lblOutput =
            new System.Windows.Forms.Label ();
         this.btnCancel =
            new System.Windows.Forms.Button ();

         // определить заголовок формы
         this.Text = "Hello World";

         // инициализация статического текста
         lblOutput.Location =
            new System.Drawing.Point (16, 24);
         lblOutput.Text = "Hello World!";
```

```
            lblOutput.Size =
               new System.Drawing.Size (216, 24);

            // инициализация кнопки Cancel
            btnCancel.Location =
               new System.Drawing.Point (150,200);
            btnCancel.Size =
               new System.Drawing.Size (112, 32);
            btnCancel.Text = "&Cancel";

            // определить обработчик события
            btnCancel.Click +=
               new System.EventHandler (this.btnCancel_Click);

            // добавить элементы в форму и установить
            // размер рабочей области
            this.AutoScaleBaseSize =
               new System.Drawing.Size (5, 13);
            this.ClientSize =
               new System.Drawing.Size (300, 300);
            this.Controls.Add (this.btnCancel);
            this.Controls.Add (this.lblOutput);

         }

         // обработать нажатие кнопки Cancel
         protected void btnCancel_Click (
            object sender, System.EventArgs e)
         {
            Application.Exit();
         }

         // запустить приложение
         public static void Main()
         {
            Application.Run(new HandDrawnClass());
         }
      }
   }
```

Использование среды разработки Visual Studio .NET

Хотя создавать программы вручную довольно интересно, это весьма трудоемкое занятие, а результат, как, например, в предыдущем случае, далеко не элегантен. Интегрированная среда разработки Visual Studio .NET предоставляет инструмент для создания форм Windows, который гораздо проще в обращении.

Чтобы начать работу над новым Windows-приложением, откройте Visual Studio .NET и вызовите окно New Project. В окне NewProject создайте новое Windows-приложение на C# и назовите его ProgCSharpWindowsForm, как показано на рис. 13.2.

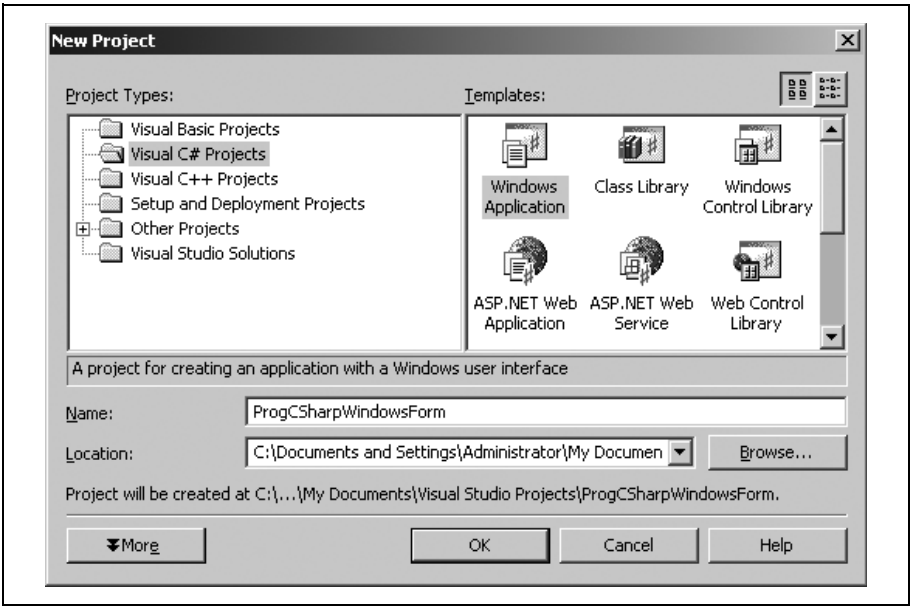

Рис. 13.2. *Создание приложения Windows Forms*

Visual Studio .NET отреагирует созданием приложения Windows Forms и, что еще важнее, переведет пользователя в среду визуальной разработки (рис. 13.3).

Окно Design содержит пустую форму Windows (с именем `Form1`). Здесь же имеется окно Toolbox, в котором можно выбирать элементы управления. Если это окно отсутствует на экране, щелкните по вкладке `Toolbox` или выберите команду меню View → Toolbox. Для этого можно воспользоваться также комбинацией клавиш <Ctrl>+<Alt>+<X>. Из окна Toolbox следует перетащить на форму элементы управления Label и Button, как показано на рис. 13.4.

Прежде чем идти дальше, оглянемся вокруг. Окно Toolbox содержит большое число элементов управления, которые можно добавлять на форму Windows-приложения. В правом верхнем углу главного окна находится окно Solution Explorer, которое содержит все файлы проектов. В правом нижнем углу главного окна расположено окно Properties. Оно содержит свойства текущего элемента. На рис. 13.4 выделен статический текст (`label1`), поэтому окно Properties содержит *его* свойства.

Это окно позволяет устанавливать статические свойства различных элементов управления. Например, чтобы добавить к элементу управления `label1` требуемый текст, следует справа от свойства `Text` ввести слова «Hello World». Если возникнет желание изменить шрифт этого текста, нужно будет щелкнуть по свойству `Font` (рис. 13.5). Аналогичным образом можно задать текст, отображаемый на кнопке (`button1`), выбрав ее в окне Properties и введя слово «Cancel» в свойстве `Text`.

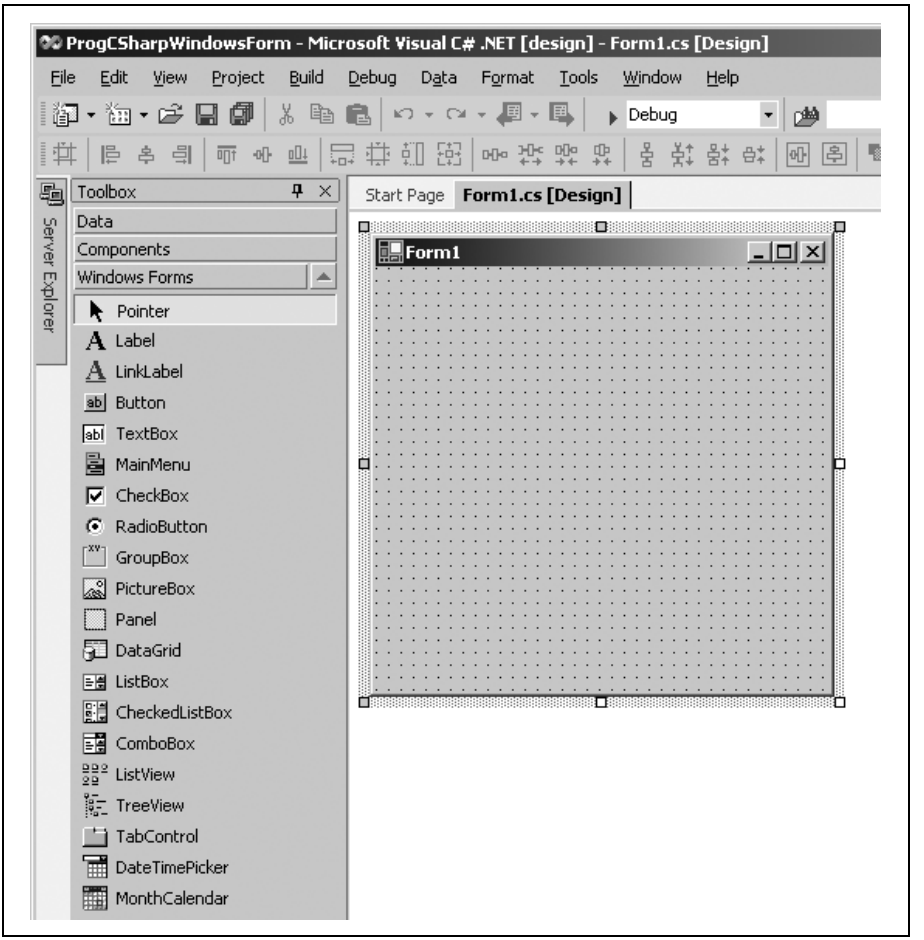

Рис. 13.3. *Среда визуальной разработки*

Любое из этих действий выполнить гораздо проще, чем вносить изменения в текст программы (хотя внесение таких изменений по-прежнему возможно).

После того как элементы управления размещены на форме, можно приступать к созданию обработчика события для кнопки Cancel. Двойной щелчок по ней приведет к созданию и регистрации обработчика события и к появлению *страницы поддержки* (страницы с исходным кодом формы), в которой можно задать логику работы обработчика события (рис. 13.6).

Курсор уже находится в нужном месте, остается лишь ввести следующую строку программы:

```
Application.Exit();
```

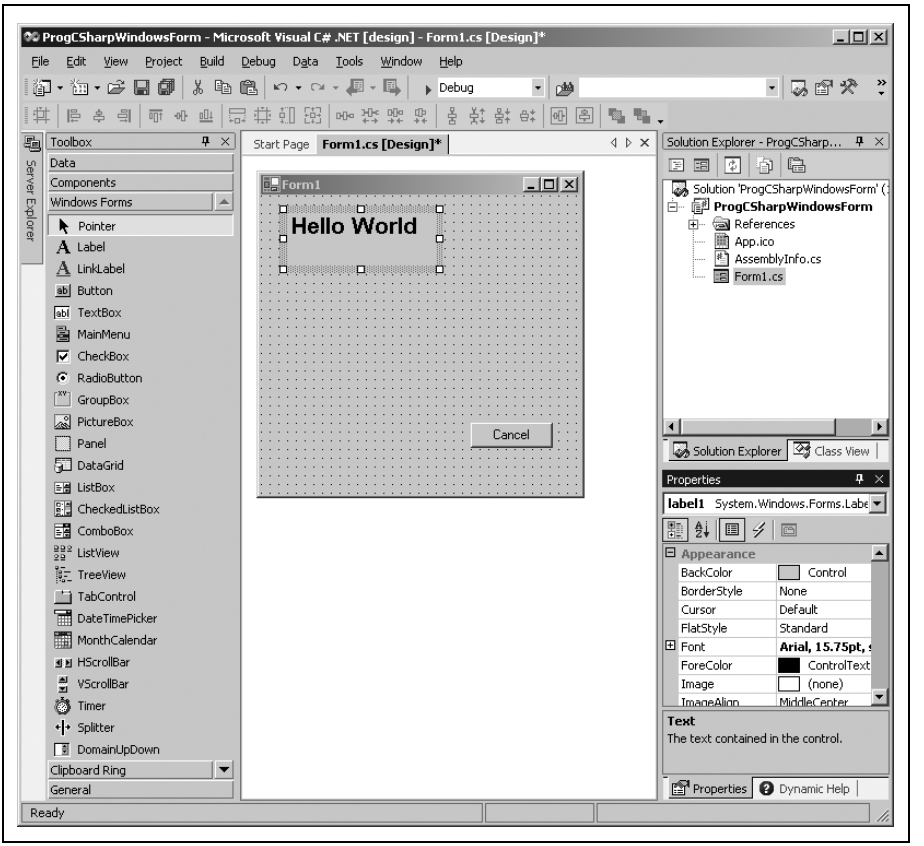

Рис. 13.4. *Среда разработки Windows Forms*

В интегрированной среде разработки Visual Studio .NET курсор мигает, что позволяет быстро понять, куда вводится текст. Для большинства читателей курсор на рисунках в книге не мигает.

Visual Studio .NET создает заготовку программы, реализующей создание и инициализацию компонентов. Ее полный исходный текст приведен в примере 13.2. Он включает в себя и строку, введенную вручную (выделена полужирным шрифтом), которая обрабатывает нажатие кнопки Cancel.

Пример 13.2. Исходный код, сгенерированный IDE[1]

```
using System;
```

[1] В этом и следующих примерах сгенерированный код содержит комментарии на английском языке, автоматически добавляемые интегрированной средой разработки. Эти комментарии оставлены без перевода. – *Примеч. науч. ред.*

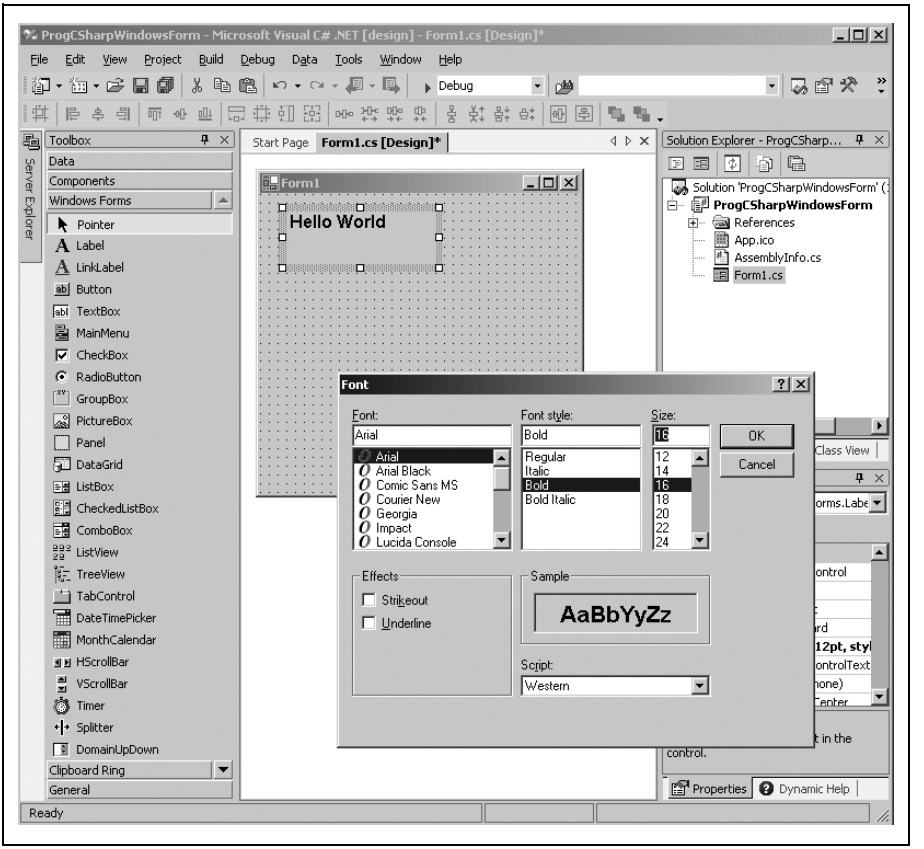

Рис. 13.5. *Изменение шрифта*

```
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace ProgCSharpWindowsForm
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
        public class Form1 : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Label lblOutput;
        private System.Windows.Forms.Button btnCancel;
        /// <summary>
        /// Required designer variable.
        /// </summary>
```

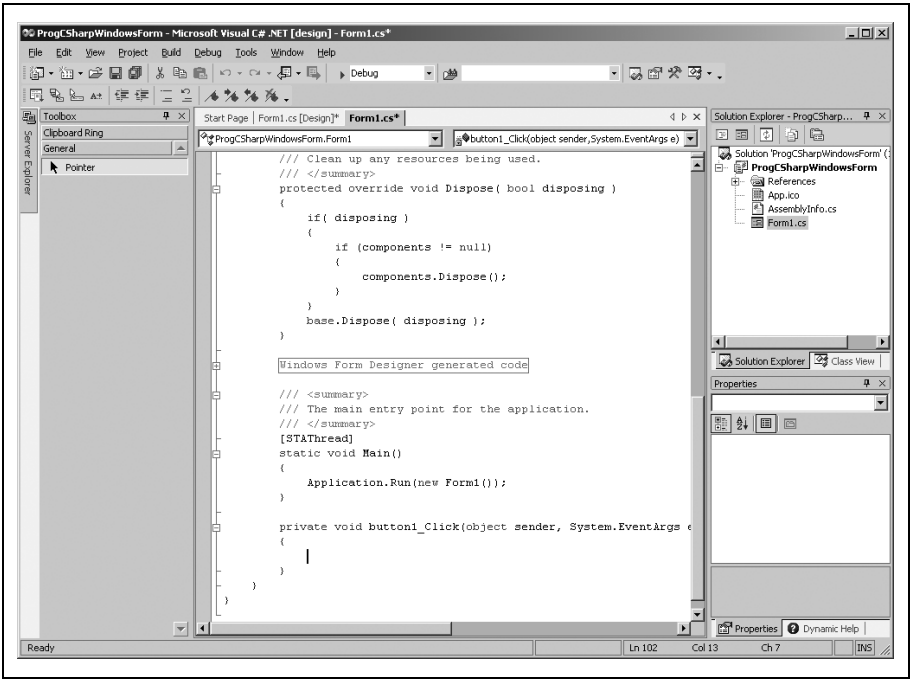

Рис. 13.6. *Результат двойного щелчка по кнопке Cancel*

```
private System.ComponentModel.Container components = null;

public Form1(  )
{
    //
    // Required for Windows Form Designer support
    //
                InitializeComponent(  );

    //
    // TODO: Add any constructor code
    // after InitializeComponent call
    //


}

/// <summary>
/// Clean up any resources being used.
/// </summary>

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if (components != null)
        {
```

```
            components.Dispose(  );
        }
    }
    base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>

private void InitializeComponent(  )
{
    this.lblOutput = new System.Windows.Forms.Label(  );
    this.btnCancel = new System.Windows.Forms.Button(  );
    this.SuspendLayout(  );
    //
    // lblOutput
    //
    this.lblOutput.Font = new System.Drawing.Font("Arial", 15.75F,
        System.Drawing.FontStyle.Bold,
        System.Drawing.GraphicsUnit.Point, ((System.Byte)(0)));
    this.lblOutput.Location = new System.Drawing.Point(24, 16);
    this.lblOutput.Name = "lblOutput";
    this.lblOutput.Size = new System.Drawing.Size(136, 48);
    this.lblOutput.TabIndex = 0;
    this.lblOutput.Text = "Hello World";
    //
    // btnCancel
    //
    this.btnCancel.Location = new System.Drawing.Point(192, 208);
    this.btnCancel.Name = "btnCancel";
    this.btnCancel.TabIndex = 1;
    this.btnCancel.Text = "Cancel";
    this.btnCancel.Click += new
    System.EventHandler(this.btnCancel_Click);
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(292, 273);
    this.Controls.AddRange(new System.Windows.Forms.Control[] {
            this.btnCancel, this.lblOutput});
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);

}
#endregion

/// <summary>
```

```
        /// The main entry point for the application.
        /// </summary>

        [STAThread]
        static void Main(  )
        {
            Application.Run(new Form1(  ));
        }
        private void btnCancel_Click(object sender, System.EventArgs e)
        {

            Application.Exit( );
        }

    }
}
```

Некоторые строчки этого примера отформатированы так, чтобы они поместились на странице книги.

В этом примере имеется код, отсутствовавший в примере 13.1, хотя этот код и не является абсолютно необходимым. Дело в том, что, создавая приложение, Visual Studio добавляет шаблонный код, не нужный в такой простой программе, как эта.

Внимательное изучение программы показывает, что в основном она аналогична предыдущей, но некоторые важные отличия заслуживают особого разговора. Пример начинается со специальных комментариев:

```
/// <summary>
/// Summary description for Form1.
/// </summary>
```

Специальный комментарий служит для документирования кода и подробно рассматривается далее в этой главе. Форма, создаваемая в программе, произведена от класса `System.Windows.Forms.Form`, как и форма из предыдущего примера. Аналогично определены и элементы управления:

```
public class Form1 : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label lblOutput;
   private System.Windows.Forms.Button btnCancel;
```

Переменная `container` создается средой программирования для внутреннего использования:

```
private System.ComponentModel.Container components;
```

В этом приложении, как и в любом другом, созданном средой Visual Studio .NET, конструктор вызывает закрытый метод `InitializeCompo-`

nent(), который служит для создания и установки свойств всех элементов управления. Свойствам присваиваются значения, установленные разработчиком при создании формы (или значения, используемые по умолчанию, если он их не изменил). Метод InitializeComponent() снабжен комментарием, предостерегающим программиста от внесения изменений в его текст. Если не послушаться этого предостережения, среда может отреагировать неадекватно.

Программа, созданная средой разработки, будет вести себя точно так же, как и программа, написанная вручную.

Создание приложения Windows Forms

Чтобы увидеть, как можно использовать Windows Forms для создания более серьезного Windows-приложения, создадим утилиту FileCopier, копирующую все файлы из группы каталогов, выделенных пользователем, в один каталог назначения или на устройство, например на дискету или на диск, выделенный в локальной сети для хранения резервных копий. Хотя здесь будут реализованы не все необходимые функциональные возможности, это приложение можно доработать так, что отмеченные пользователем файлы будут копироваться на несколько дисков и размещаться там как можно плотнее. Более того, можно добавить в приложение функцию сжатия файлов. Истинная же цель этого примера состоит в том, чтобы читатель поупражнялся в применении приемов программирования на C#, описанных в предыдущих главах, и исследовал пространство имен Windows.Forms.

По этой причине, а также ради простоты программы все внимание будет сосредоточено на пользовательском интерфейсе и на включении в него различных элементов управления. Окончательный вид создаваемого приложения представлен на рис. 13.7.

Пользовательский интерфейс программы FileCopier состоит из следующих элементов управления:

- Статический текст: Source Files и Target Directory
- Кнопки: Clear, Copy, Delete и Cancel
- Флажок Overwrite if exists
- Текстовое поле, содержащее путь к каталогу назначения
- Два больших окна древовидных списков: одно для исходных каталогов, другое для устройств и каталогов назначения

Такой интерфейс приложения позволяет пользователю выделять файлы (или целые каталоги) в левом окне с древовидным списком (источнике). После щелчка по кнопке Copy файлы, отмеченные в левом окне, будут скопированы в каталог, указанный в правом элементе управления. Если пользователь нажмет кнопку Delete, выделенные файлы будут удалены.

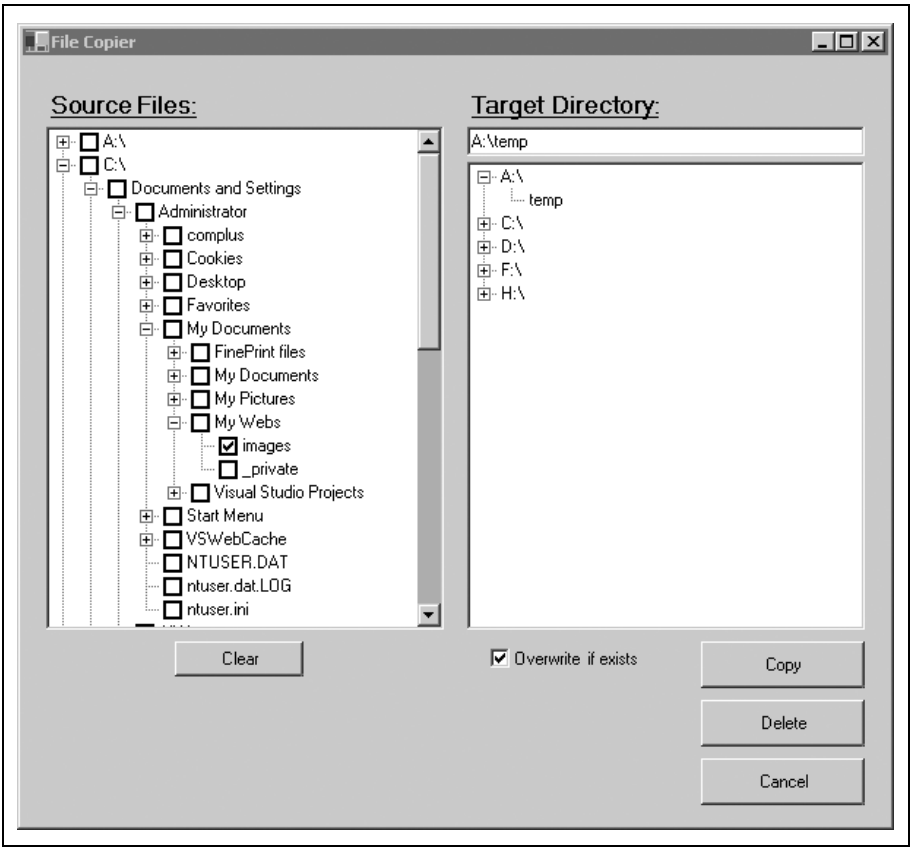

Рис. 13.7. *Пользовательский интерфейс приложения FileCopier*

В последующих разделах этой главы будут реализованы функциональные возможности приложения `FileCopier`, что позволит продемонстрировать технику работы в среде Windows Forms.

Создание базовой формы пользовательского интерфейса

В первую очередь необходимо открыть новый проект и назвать его `FileCopier`. Интегрированная среда разработки Visual Studio .NET перейдет в режим визуальной разработки, где можно будет перетаскивать элементы прямо на форму. Размер формы можно изменять по желанию. Из окна Toolbox перетащите в форму следующие элементы и установите их свойство `Name`: статические тексты (`lblSource`, `lblTarget`, `lblStatus`), кнопки (`btnClear`, `btnCopy`, `btnDelete`, `btnCancel`), флажок (`chkOverwrite`), текстовое поле (`txtTargetDir`) и древовидные списки (`tvwSource`, `tvwTargetDir`). Постарайтесь, чтобы форма выглядела, как показано на рис. 13.8.

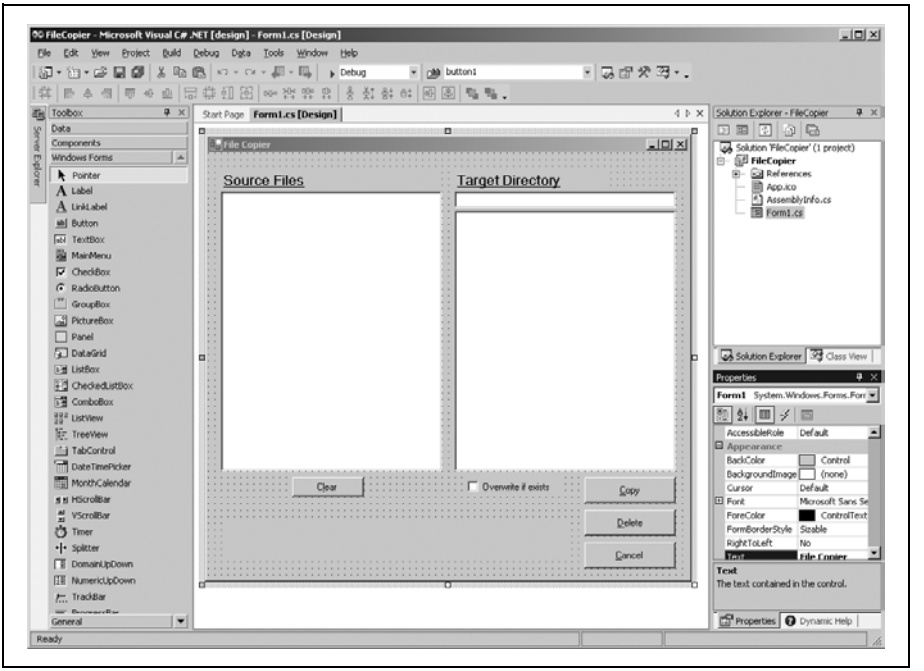

Рис. 13.8. *Создание формы в режиме визуального проектирования*

В левом окне древовидного списка требуется устанавливать флажки рядом с именами выделенных каталогов и файлов, а в правом окне (где указывается только один каталог) это не требуется. Для левого окна древовидного списка (`tvwSource`) установите значение `true` для свойства `CheckBoxes`, а для правого окна (`tvwTargetDir`) установите для данного свойства значение `false`. Чтобы сделать это, щелкните по каждому из этих элементов управления и установите соответствующее значение в окне Properties.

Закончив установку свойств, дважды щелкните по кнопке Cancel, чтобы создать обработчик события для нее. Вообще, в ответ на двойной щелчок по элементу управления среда Visual Studio .NET создает обработчик события для соответствующего объекта. В данном случае событие ожидается только одно, и Visual Studio .NET создает обработчик этого события:

```
protected void btnCancel_Click (object sender, System.EventArgs e)
{
   Application.Exit();
}
```

С окном древовидного списка связано много разных событий. Для создания соответствующих обработчиков следует нажать кнопку Events в окне Properties. После этого можно создавать новые обработчики собы-

тий, выделяя их имена в раскрывающихся списках, связанных с именами обрабатываемых событий. Visual Studio .NET зарегистрирует обработчик и откроет в текстовом редакторе созданную заготовку метода, поместив курсор в пустое тело обработчика.

На этом автоматическое создание заготовки программы заканчивается. Visual Studio .NET создает форму и инициализирует все элементы управления, однако не заполняет окна древовидных списков. Эту операцию приходится проделать вручную.

Заполнение окна иерархического списка

В создаваемом приложении оба элемента управления TreeView функционируют практически одинаково, но левый, tvwSource, содержит каталоги и файлы, а правый, tvwTargetDir, – только каталоги. У объекта tvwSource свойство CheckBoxes имеет значение true, а у tvwTargetDir оно имеет значение false. Кроме того, хотя объект tvwSource разрешает одновременное выделение нескольких элементов списка (это поведение определено для объектов его класса по умолчанию), в объекте tvwTargetDir множественное выделение должно быть запрещено.

Выделим общие для обоих объектов класса TreeView операции в совместно используемый метод FillDirectoryTree(), которому будем передавать объект и флажок, показывающий, нужно ли получать список файлов. Этот метод будет вызван в конструкторе для каждого из этих двух элементов управления:

```
FillDirectoryTree(tvwSource, true);
FillDirectoryTree(tvwTargetDir, false);
```

В реализации метода FillDirectoryTree()параметр типа TreeView называется tvw. Он будет представлять то один, то другой объект TreeView. Вам потребуются методы из пространства имен System.IO, поэтому добавим оператор using System.IO; в начало файла Form1.cs. Затем добавим в файл Form1.cs объявление метода:

```
private void FillDirectoryTree(TreeView tvw, bool isSource)
```

Объекты TreeNode

Объект TreeView имеет свойство Nodes, возвращающее объект TreeNodeCollection. Этот объект представляет собой коллекцию объектов класса TreeNode, каждый из которых представляет собой узел древовидной структуры. Начнем с того, что очистим эту коллекцию:

```
tvw.Nodes.Clear();
```

Теперь можно заполнять свойство Nodes объекта TreeView, рекурсивно обходя каталоги всех дисков. Для начала получим имена всех логических дисков системы. Для этого воспользуемся статическим методом GetLogicalDrives() объекта Environment. Класс Environment предоставляет

программисту информацию о текущем окружении на данной платформе. Иначе говоря, с помощью объекта класса Environment можно узнать название компьютера, на котором выполняется программа, версию ОС, системный каталог и т. д.

```
string[] strDrives = Environment.GetLogicalDrives();
```

Метод GetLogicalDrives() возвращает массив строк, каждая из которых представляет корневой каталог одного из логических дисков. Перебирая в цикле элементы этого массива, можно добавлять узлы в объект TreeView.

```
foreach (string rootDirectoryName in strDrives)
{
```

В цикле foreach должен быть обработан каждый диск. Чтобы ограничить область поиска одним диском (это полезно, если у вас несколько дисков большого объема или подключены сетевые диски), добавьте следующие две строки:

```
if (rootDirectoryName != @"C:\")
continue
```

В первую очередь необходимо убедиться, что диск готов к обмену информацией. С этой целью применяется следующий «хитрый» прием. Получим список каталогов верхнего уровня на данном диске, вызвав метод GetDirectories() объекта DirectoryInfo, созданного для корневого каталога:

```
DirectoryInfo dir = new DirectoryInfo(rootDirectoryName);
dir.GetDirectories();
```

Класс DirectoryInfo предоставляет программисту методы для создания, перемещения и перебора каталогов, файлов и подкаталогов. Этот класс подробно описан в главе 21.

Метод GetDirectories() возвращает список каталогов, но в этой программе он абсолютно не нужен. Метод вызывается только ради того, чтобы было вызвано исключение, если диск не готов к работе.

Вызов метода GetDirectories() помещается в блок try, а в блоке catch не предпринимаются никакие действия. В результате при вызове исключения диск просто игнорируется.

Если же диск находится в состоянии готовности, создадим объект TreeNode, который будет содержать корневой каталог данного диска, и добавим этот узел в иерархический список TreeView:

```
TreeNode ndRoot = new TreeNode(rootDirectoryName);
tvw.Nodes.Add(ndRoot);
```

Чтобы рекурсивно просмотреть все каталоги, нужно вызвать новый для читателя метод GetSubDirectoryNodes(), передав ему корневой узел,

имя корневого каталога и флажок, показывающий, требуется ли выводить файлы:

```
if (isSource)
{
   GetSubDirectoryNodes(ndRoot, ndRoot.Text, true);
}
else
{
   GetSubDirectoryNodes(ndRoot, ndRoot.Text, false);
}
```

Читатель, возможно, задается вопросом, зачем передавать `ndRoot.Text` вместе с `ndRoot`. Минутку терпения, все станет понятно после рекурсивного вызова метода `GetSubDirectoryNodes()`. На этом метод `FillDirectoryTree()` заканчивается. Его полный текст приведен в примере 13.3.

Рекурсивный просмотр каталогов

Метод `GetSubDirectoryNodes()` начинается с вызова метода `GetDirectories()` и сохранения возвращенного им массива объектов `DirectoryInfo`:

```
private void GetSubDirectoryNodes(
   TreeNode parentNode, string fullName, bool getFileNames)
{
   DirectoryInfo dir = new DirectoryInfo(fullName);
   DirectoryInfo[] dirSubs = dir.GetDirectories();
```

Обратите внимание, что узел, передаваемый методу, называется `parentNode` (родительский узел). Узлы текущего уровня будут считаться потомками узла, переданного в качестве аргумента. Так структура каталогов будет отображаться в иерархическую структуру элемента управления.

Просмотрим в цикле все подкаталоги, игнорируя скрытые:

```
foreach (Directory dirSub in dirSubs)
{
    if ( (dirSub.Attributes &
         FileSystemAttributes.Hidden) != 0 )
    {
        continue;
    }
```

`FileSystemAttributes` — это перечисление, элементами которого могут быть `Archive`, `Compressed`, `Directory`, `Encrypted`, `Hidden`, `Normal`, `ReadOnly` и др.

Свойство `dirSub.Attributes` является битовым шаблоном текущих атрибутов каталога. Когда выполняется операция «логическое И» над этим значением и битовым шаблоном `FileSystemAttributes.Hidden`, бит устанавливается, если файл является скрытым (имеет атрибут `hidden`); в противном случае все биты сбрасываются. Сравнение результата (имеющего тип `int`) с нулем позволяет узнать значение атрибута.

Далее создается объект TreeNode с именем данного каталога, и этот объект добавляется в коллекцию Nodes узла, переданного методу в качестве параметра (parentNode):

```
TreeNode subNode = new TreeNode(dirSub.Name);
parentNode.Nodes.Add(subNode);
```

Теперь необходимо выполнить рекурсивный вызов метода GetSubDirectoryNodes(), передав только что созданный узел в качестве родительского, полный путь в качестве имени родительского узла, а также флаг – третий параметр:

```
GetSubDirectoryNodes(subNode,dirSub.FullName,getFileNames);
```

Обратите внимание, что в вызове конструктора TreeNode использовано свойство Name объекта DirectoryInfo, а в вызове метода GetSubDirectoryNodes() использовано свойство FullName. Дело в том, что если полным именем каталога является, например, c:\WinNT\Media\Sounds, то свойство FullName возвратит это имя целиком, а свойство Name возвратит только Sounds. Конструктору узла передается только имя, поскольку оно должно фигурировать в древовидной структуре на экране. Полное имя передается методу GetSubDirectoryNodes(), чтобы он мог обнаружить все подкаталоги. Вот и ответ на вопрос, зачем нужно было передавать имя корневого узла при первом вызове метода: на самом деле передается не имя узла, а полный путь к каталогу, представляемому этим узлом!

Получение списка файлов в каталоге

После выполнения рекурсивного просмотра подкаталогов самое время получить список файлов в каждом из них, если, конечно, флаг getFileNames имеет значение true. Для этого следует вызвать метод GetFiles() объекта DirectoryInfo. Метод возвращает массив объектов FileInfo:

```
if (getFileNames)
{
   // получить информацию о файлах данного узла
   FileInfo[] files = dir.GetFiles();
```

Класс FileInfo (обсуждаемый в главе 21) обладает рядом методов, позволяющих работать с файлами.

Сейчас можно в цикле просмотреть элементы этой коллекции, читая свойство Name каждого объекта FileInfo и передавая это имя конструктору объекта TreeNode. После чего созданный объект добавляется в коллекцию Nodes родительского узла (в результате чего появляется еще один дочерний узел). В этом случае рекурсия отсутствует, поскольку у файлов нет подкаталогов:

```
foreach (FileInfo file in files)
{
```

```
        TreeNode fileNode = new TreeNode(file.Name);
        parentNode.Nodes.Add(fileNode);
    }
```

Заполнение иерархической структуры на этом заканчивается. Полный исходный текст этого метода можно увидеть в примере 13.3.

Если для читателя что-то осталось непонятным, автор рекомендует выполнить прорамму в отладчике в пошаговом режиме, наблюдая, как объект TreeView строит свои узлы.

Обработка событий объекта TreeView

В рассматриваемом примере необходимо обработать целый ряд событий. Во-первых, пользователь может щелкнуть по любой из кнопок: Cancel, Copy, Clear или Delete. Во-вторых, он может щелкнуть по одному из флажков в левом древовидном списке или одному из каталогов в правом.

Сначала рассмотрим работу с объектами класса TreeView, поскольку это интереснее и, возможно, труднее.

Работа с левым окном древовидного списка

В создаваемой программе существует два объекта класса TreeView, и у каждого есть свой обработчик событий. Вначале рассмотрим объект-источник. Пользователь отмечает в нем файлы и каталоги, которые он хочет скопировать. Каждый раз, когда пользователь щелкает левой кнопкой мыши по файлу или каталогу, возникает несколько событий. Событие, которое мы должны обработать, — AfterCheck.

Для его обработки создадим метод и назовем его tvwSource_AfterCheck(). Среда Visual Studio.NET свяжет его со стандартным обработчиком, а если программист ее не использует, он должен будет сделать это вручную:

```
tvwSource.AfterCheck +=
new System.Windows.Forms.TreeViewEventHandler (this.tvwSource_AfterCheck);
```

Реализация метода AfterCheck() делегирует работу рекурсивному методу SetCheck(), который тоже придется написать самостоятельно. Чтобы добавить обработчик события AfterCheck, выделите элемент управления tvwSourse, щелкните по значку Events в окне Properties, затем дважды щелкните по элементу AfterCheck. После этого будет добавлено событие и в окне редактирования откроется исходный код обработчика, где можно будет ввести тело метода.

```
private void tvwSource_AfterCheck (
object sender, System.Windows.Forms.TreeViewEventArgs e)
{
    SetCheck(e.Node,e.Node.Checked);
}
```

Обработчик события принимает объект sender и объект типа TreeViewEventArgs. Оказывается, узел вполне доступен из объекта этого типа (по имени e). Методу SetCheck() передадим узел и его свойство, показывающее, выделен ли объект пользователем.

Каждый объект node обладает свойством Nodes, которое возвращает коллекцию TreeNodeCollection, содержащую все дочерние узлы. Метод SetCheck() рекурсивно просматривает коллекцию Nodes текущего узла, отмечая каждый дочерний, если отмечен его родитель. Иными словами, если отмечен каталог, то все его файлы и подкаталоги рекурсивно отмечаются, сверху вниз.

Там черепахи до самого низа

Когда речь заходит о рекурсии, автор этих строк вспоминает одну историю. Как-то раз известный дарвинист выступал с лекцией, развенчивающей мифы о создании мира. «Некоторые, – говорил он, – верят, что мир покоится на спине огромной черепахи. Естественно, возникает вопрос, на чем стоит эта черепаха?»

С последнего ряда поднялась пожилая дама и сказала: «Очень умный вопрос, сынок, но там черепахи до самого низа».

Для каждого объекта класса TreeNode из коллекции Nodes выполняется проверка, не является ли он листом дерева. Узел является листом, если свойство Count его коллекции Nodes равно нулю. Если это так, свойству Checked присваивается значение переданного параметра; в противном случае выполняется рекурсия:

```
private void SetCheck(TreeNode node, bool check)
{
    // найти все дочерние узлы текущего узла
    foreach (TreeNode n in node.Nodes)
    {
       n.Checked = check;   // проверка узла

       // если это тоже узел дерева - рекурсия
      if (n.Nodes.Count != 0)
       {
          SetCheck(n,check);
       }
    }
}
```

Таким образом выделение (или его отсутствие) распространяется вниз по всей структуре. Пользователю достаточно выделить каталог, чтобы показать, что он хочет выделить все файлы во всех его подкаталогах.

Работа с правым окном иерархического списка

Обработчик событий для объекта назначения класса TreeView несколько сложнее. Само событие носит имя AfterSelect. (Вспомним, что в этом древовидном списке нет флажков.) На этот раз необходимо поместить в текстовое поле (над правой древовидной структурой) полный путь к каталогу, отмеченному пользователем.

В этой целью необходимо пройти по всем узлам, находя для каждого из них имя родительского каталога и строя полный путь:

```
private void tvwTargetDir_AfterSelect (
    object sender, System.Windows.Forms.TreeViewEventArgs e)
{
    string theFullPath = GetParentString(e.Node);
```

Метод GetParentString() будет рассмотрен чуть ниже. Когда полный путь получен, необходимо удалить символ обратной наклонной черты в его конце (если таковой имеется), и после этого можно выводить путь в текстовое поле:

```
if (theFullPath.EndsWith("\\"))
{
    theFullPath = theFullPath.Substring(0,theFullPath.Length-1);
}
txtTargetDir.Text = theFullPath;
```

Метод GetParentString() принимает узел и возвращает строку с полным путем к нему. С этой целью метод выполняет рекурсию вверх по дереву каталогов, ставя обратную наклонную черту после каждого узла, не являющегося листом дерева:

```
private string GetParentString(TreeNode node)
{
    if(node.Parent == null)
    {
        return node.Text;
    }
    else
    {
        return GetParentString(node.Parent) + node.Text +
            (node.Nodes.Count == 0  ? "" : "\\");
    }
}
```

Условная операция (?) является единственной тернарной операцией в языке C# (тернарной операцией называется операция с тремя операндами). Ее семантика такова: «Проверить node.Nodes.Count на равенство нулю; в случае положительного результата возвратить значение, стоящее до двоеточия (в данном выражении пустую строку); иначе – возвратить значение, стоящее после двоеточия (обратную наклонную черту)».

Рекурсия прекращается, когда родительский узел отсутствует, то есть метод дошел до корневого каталога.

Обработка нажатия кнопки Clear

Когда у программиста есть метод SetCheck(), созданный в предыдущем подразделе, обработка нажатия кнопки Clear становится тривиальной:

```
private void btnClear_Click (object sender, System.EventArgs e)
{
     foreach (TreeNode node in tvwSource.Nodes)
     {
         SetCheck(node, false);
     }
}
```

Вызываем метод SetCheck() для корневых узлов и сообщаем ему, что нужно рекурсивно сбросить флажки у всех дочерних узлов.

Обработка нажатия кнопки Copy

Теперь, когда приложение умеет выделять файлы и определять каталог назначения, настало время обработать нажатие кнопки Copy. Первое, что требуется сделать, – это получить список выделенных файлов. Нужен массив объектов FileInfo, но их количество заранее неизвестно. В такой ситуации нет ничего лучше, чем объект класса ArrayList. Ответственность за заполнение списка файлов будет делегирована методу GetFileList():

```
protected void btnCopy_Click (
       object sender, System.EventArgs e)
{
    ArrayList fileList = GetFileList();
```

Рассмотрим этот метод более подробно.

Получение списка выделенных файлов

Начнем с создания объекта класса ArrayList, в котором будут храниться строки с именами файлов, отмеченных пользователем:

```
private ArrayList GetFileList()
{
    ArrayList fileNames = new ArrayList();
```

Чтобы получить эти имена, необходимо просмотреть иерархический список TreeView:

```
foreach (TreeNode theNode in tvwSource.Nodes)
{
    GetCheckedFiles(theNode, fileNames);
}
```

Чтобы узнать, как работает этот фрагмент программы, необходимо разобраться с методом GetCheckedFiles(). Сам метод предельно прост: если у переданного ему узла нет потомков (node.Nodes.Count == 0), значит он представляет собой лист дерева. Если лист дерева отмечен пользователем, строится полный путь к нему (с помощью метода GetParentString()), который и добавляется в список ArrayList, переданный в качестве второго аргумента:

```
private void GetCheckedFiles(TreeNode node,
       ArrayList fileNames)
{
    if (node.Nodes.Count == 0)
    {
        if (node.Checked)
        {
             string fullPath = GetParentString(node);
             fileNames.Add(fullPath);
        }
    }
```

Если же узел *не* является листом, выполняется рекурсия вниз по иерархическому списку для поиска дочерних узлов:

```
    else
    {
        foreach (TreeNode n in node.Nodes)
        {
            GetCheckedFiles(n,fileNames);
        }
    }
}
```

В результате получается объект класса ArrayList, заполненный именами копируемых файлов. По возвращении в метод GetFileList() этот список будет использован для создания второго объекта того же типа, предназначенного для хранения объектов FileInfo:

```
ArrayList fileList = new ArrayList();
```

Обратите внимание, что конструктор ArrayList() снова вызывается без аргументов. В этом проявляется одно из преимуществ иерархической системы классов: коллекции достаточно знать, что она содержит объекты типа Object, от которого произведены все классы. Список может хранить объекты FileInfo с такой же легкостью, как, например, объекты типа string.

Теперь все элементы списка ArrayList можно просмотреть в цикле, поочередно выбирая каждое имя и создавая с его помощью объект класса FileInfo. Чтобы выяснить, файл это или каталог, следует проанализировать свойство Exists, которое возвращает false, если объект, созданный конструктором File(), на самом деле является каталогом. Если же это действительно файл, его можно смело добавлять в список ArrayList:

```
foreach (string fileName in fileNames)
{
    FileInfo file = new FileInfo(fileName);
    if (file.Exists)
    {
        fileList.Add(file);
    }
}
```

Сортировка списка выделенных файлов

Было бы разумно копировать выделенные файлы в порядке убывания их размеров, чтобы они располагались на принимающем диске как можно более плотно. Следовательно, необходимо отсортировать список ArrayList. Можно было бы вызвать метод Sort(), но откуда взять критерии сортировки объектов типа File? Ведь объект ArrayList не имеет информации о своем содержимом.

Чтобы решить эту проблему, необходимо передать методу Sort() интерфейс IComparer. Создадим класс FileComparer, который реализует этот интерфейс и знает, как сортировать объекты FileInfo:

```
public class FileComparer : IComparer
{
```

У этого класса будет только один метод, Compare(), который принимает два объекта в качестве аргументов:

```
public int Compare (object f1, object f2)
{
```

При нормальном подходе следует возвращать 1, если первый объект (f1) больше второго (f2), -1, если верно обратное, и 0, когда объекты равны. Однако в данном случае список нужно отсортировать в порядке убывания, и логика должна быть инвертирована.

Поскольку такая реализация метода Compare() больше нигде не используется, эту необычную логику (сортировка от большего к меньшему) можно спрятать внутри метода. Альтернативное решение будет состоять в традиционной сортировке и переворачивании результатов уже в вызвавшем методе, что можно было видеть в примере 12.1.

Чтобы узнать размер объекта FileInfo, нужно привести параметры, имеющие тип Object, к типу FileInfo (что вполне безопасно, поскольку данный метод ничего другого не получит):

```
FileInfo file1 = (FileInfo) f1;
FileInfo file2 = (FileInfo) f2;
if (file1.Length > file2.Length)
{
    return -1;
```

```
        }
        if (file1.Length < file2.Length)
        {
           return 1;
        }
        return 0;
    }
}
```

В реальной программе было бы разумно проверять тип сравниваемых объектов и, возможно, обрабатывать исключение, если объекты имеют не тот тип, который ожидается.

Но вернемся к методу `GetFileList()`. Создадим ссылку на объект `IComparer` и передадим ее методу `Sort()`объекта `fileList`:

```
IComparer comparer = (IComparer) new FileComparer();
fileList.Sort(comparer);
```

Теперь можно возвращать `fileList` вызвавшему методу:

```
return fileList;
```

Вызвавшим методом был `btnCopy_Click()`. Вспомним, что метод `GetFileList()` был вызван в первой строке обработчика события!

```
protected void btnCopy_Click (object sender, System.EventArgs e)
{
    ArrayList fileList = GetFileList();
```

В этой точке метод `GetFileList()` возвращает отсортированный список объектов `File`, каждый из которых представляет файл, выделенный в исходном иерархическом списке `TreeView`.

Теперь элементы списка можно перебрать в цикле, копируя файлы и обновляя элементы пользовательского интерфейса:

```
foreach (FileInfo file in fileList)
{
   try
   {
        lblStatus.Text = "Copying " +
          txtTargetDir.Text + "\\" +
            file.Name + "...";
        Application.DoEvents();

        file.CopyTo(txtTargetDir.Text + "\\" +
            file.Name,chkOverwrite.Checked);
    }
   catch (Exception ex)
   {
        MessageBox.Show(ex.Message);
   }
}
```

```
lblStatus.Text = "Done.";
Application.DoEvents();
```

В ходе этой процедуры в переменную lblStatus записывается текст, сообщающий о выполняемой операции копирования, а затем вызывается метод Application.DoEvents() для вывода его на экран. После этого для данного файла вызывается метод CopyTo(), которому передается принимающий каталог, имя которого получено из текстового поля, и флаг, определяющий необходимость обновлять существующие файлы.

Нетрудно заметить, что значение этого флага отражает состояние флажка chkOverWrite. Свойство Checked этого объекта равно true, если флажок установлен, и false в противном случае.

Копирование происходит внутри блока try, поскольку является операцией, в ходе которой возможны всякие неприятности. В данной программе обработка всех исключений сводится к выводу на экран диалогового окна с сообщением об ошибке. В коммерческом приложении, скорее всего, были бы предприняты корректирующие действия.

Вот и все. Копирование реализовано!

Обработка событий кнопки Delete

Метод обработки сообщения о щелчке по кнопке Delete еще проще. Во-первых, нужно попросить пользователя подтвердить свое желание удалить файлы:

```
protected void btnDelete_Click
(object sender, System.EventArgs e)
{
System.Windows.Forms.DialogResult result =
   MessageBox.Show(
   "Are you quite sure?",              // сообщение
   "Delete Files",                     // заголовок
   MessageBoxButtons.OKCancel,         // кнопки
   MessageBoxIcon.Exclamation,         // значок
   MessageBoxDefaultButton.Button2);   // кнопка по умолчанию
```

Здесь использован статический метод Show() класса MessageBox, которому передается выводимое сообщение («Are you quite sure?»), заголовок окна («Delete Files») и флаги:

- MessageBox.OKCancel требует, чтобы окно содержало две кнопки, OK и Cancel.
- MessageBox.IconExclamation означает, что в окне должен присутствовать значок, содержащий треугольник с восклицательным знаком.
- MessageBox.DefaultButton.Button2 при выводе диалоговго окна передает фокус ввода второй кнопке (Cancel).

Когда пользователь нажимает кнопку OK или Cancel, диалоговое окно возвращает значение перечислимого типа System.Windows.Forms.Dialog-

Result. В следующем фрагменте программы проверяется, щелкнул ли пользователь по кнопке OK:

```
if (result == System.Windows.Forms.DialogResult.OK)
{
```

Если ответ утвердительный, можно получить список имен (fileNames) и просмотреть его элементы в цикле, удаляя соответствующие файлы:

```
ArrayList fileNames = GetFileList();

foreach (FileInfo file in fileNames)
{
   try
   {
        lblStatus.Text = "Deleting " +
            txtTargetDir.Text + "\\" +
            file.Name + "...";
        Application.DoEvents();

        file.Delete();
    }

   catch (Exception ex)
   {
        MessageBox.Show(ex.Message);
   }
}
lblStatus.Text = "Done.";
Application.DoEvents();
```

Этот фрагмент программы аналогичен фрагменту программы, выполняющему копирование, но в данном случае вызывается метод Delete().

В примере 13.3 приводится текст программы обсуждаемого приложения, снабженный комментариями.

В целях экономии места в книге представлены только методы, написанные автором. Опущены объявления объектов Windows.Forms и заготовки, созданные средой Visual Studio .NET. Как было сказано в предисловии, читатель может загрузить полный текст программы с веб-сайта автора *http://www.LibertyAssociates.com*.

Пример 13.3. Исходный код программы FileCopier

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
```

```
/// <remarks>
///    File Copier - демонстрационная программа WinForms
///    (c) Copyright 2001 Liberty Associates, Inc.
/// </remarks>
namespace FileCopier
{
    /// <summary>
    /// Форма, демонстрирующая реализацию Windows Forms.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {

    // < отсюда удалены объявления элементов управления Windows >

        /// <summary>
        /// Переменная, необходимая для конструктора форм
        /// </summary>
        private System.ComponentModel.Container components = null;

       /// <summary>
       ///    Внутренний класс, который умеет сравнивать два файла,
       ///    сортируемые в порядке убывания размера. Обычная
       ///    логика возвращаемых значений инвертирована.
       /// </summary>
         public class FileComparer : IComparer
        {
            public int Compare (object f1, object f2)
            {
                FileInfo file1 = (FileInfo) f1;
                FileInfo file2 = (FileInfo) f2;
                if (file1.Length > file2.Length)
                {
                    return -1;
                }
                if (file1.Length < file2.Length)
                {
                    return 1;
                }
                return 0;
            }
        }

        public Form1(  )
        {
            //
            // Необходимо для поддержки конструктора Windows-форм.
            //
            InitializeComponent(  );

            // Заполнить исходный и принимающий иерархические списки.
            FillDirectoryTree(tvwSource, true);
            FillDirectoryTree(tvwTargetDir, false);
        }
```

```
/// <summary>
/// Заполнить список каталогов
/// исходного или принимающего объекта класса TreeView.
/// </summary>
private void FillDirectoryTree(
  TreeView tvw, bool isSource)
{

    //  Записать в исходный объект TreeView (tvwSource)
    //  содержание локального жесткого диска.
    //  Вначале очистить все узлы.
    tvw.Nodes.Clear(  );

    //  Прочитать имена логических дисков и поместить их
    //  в корневые узлы. Заполнить массив именами всех
    //  логических дисков компьютера.

    string[] strDrives = Environment.GetLogicalDrives(  );

    //  Просмотреть диски в цикле, добавляя их в иерархический список.
    //  Воспользоваться блоком try/catch, чтобы в случае
    //  неготовности диска (например, не вставлена дискета
    //  или CD) узел не добавлялся.
    foreach (string rootDirectoryName in strDrives)
    {
        if (rootDirectoryName != @"C:\")
            continue;
        try
        {

            //  Заполнить массив подкаталогами первого
            //  уровня. Если диск не готов,
            //  будет вызвано исключение.
            DirectoryInfo dir = new DirectoryInfo(rootDirectoryName);
            dir.GetDirectories(  );

            TreeNode ndRoot = new TreeNode(rootDirectoryName);

            //  Добавить узлы, соответствующие корневым каталогам.
            tvw.Nodes.Add(ndRoot);

            //  Добавить узлы, соответствующие подкаталогам.
            //  В случае исходного объекта TreeView
            //  получить имена файлов.
            if (isSource)
            {
                GetSubDirectoryNodes(
                  ndRoot, ndRoot.Text, true);
            }
            else
            {
                GetSubDirectoryNodes(
                  ndRoot, ndRoot.Text, false);
            }
        }
```

```
            //  Обработать все исключения,
            //  например "диск не готов".
            catch (Exception e)
            {
                MessageBox.Show(e.Message);
            }
        }
    } //  Закрывающая скобка для FillSourceDirectoryTree.

    /// <summary>
    /// Получает все подкаталоги переданного каталога.
    /// Добавляет их в иерархический список каталогов.
    /// Принимает параметры: родительский узел текущего
    /// подкаталога, полное имя этого подкаталога и
    /// флаг - индикатор необходимости получать имена
    /// файлов этого каталога.
    /// </summary>
    private void GetSubDirectoryNodes(
        TreeNode parentNode, string fullName, bool getFileNames)
    {
        DirectoryInfo dir = new DirectoryInfo(fullName);
        DirectoryInfo[] dirSubs = dir.GetDirectories(  );

        //  Добавить дочерние узлы, соответствующие подкаталогам.
        foreach (DirectoryInfo dirSub in dirSubs)
        {

            // Не показывать скрытые папки.
            if ( (dirSub.Attributes & FileAttributes.Hidden)
               != 0 )
            {
                continue;
            }

            /// <summary>
            ///    Каждое имя каталога является полным путем.
            ///    Необходимо разбить его по обратным
            ///    слэшам и воспользоваться
            ///    лишь последним узлом.
            ///    Обратные слэши должны быть удвоены,
            ///    так как в нормальной ситуации
            ///    это управляющая последовательность.
            /// </summary>
            TreeNode subNode = new TreeNode(dirSub.Name);
            parentNode.Nodes.Add(subNode);

            //  Рекурсивный вызов метода GetSubDirectoryNodes
            GetSubDirectoryNodes(
              subNode,dirSub.FullName,getFileNames);

        }
        if (getFileNames)
        {
            //  Получить файлы для данного узла
```

```
            FileInfo[] files = dir.GetFiles(  );

            // После размещения узлов
            // разместить файлы в подкаталогах.
            foreach (FileInfo file in files)
            {
                TreeNode fileNode = new TreeNode(file.Name);
                parentNode.Nodes.Add(fileNode);
            }
        }
    }

// < отсюда удалены заготовки >

    /// <summary>
    /// Главная точка входа в приложение.
    /// </summary>
    [STAThread]
    static void Main(  )
    {
        Application.Run(new Form1(  ));
    }

    /// <summary>
    ///    Создать упорядоченный список всех файлов,
    ///    выделенных пользователем, и скопировать их
    ///    в принимающий каталог.
    /// </summary>
    private void btnCopy_Click(object sender, System.EventArgs e)
    {
        // Получить список.
        ArrayList fileList = GetFileList(  );

        // Скопировать файлы.
        foreach (FileInfo file in fileList)
        {
            try
            {
                // Вывести строку состояния.
                lblStatus.Text = "Copying " + txtTargetDir.Text +
                    "\\" + file.Name + "...";
                Application.DoEvents(  );

                // Скопировать файл в место назначения.
                file.CopyTo(txtTargetDir.Text + "\\" +
                    file.Name,chkOverwrite.Checked);
            }

            catch (Exception ex)
            {
                // Возможно, читатель захочет реализовать более сложную
                // обработку исключения, чем вывод сообщения на экран.
                MessageBox.Show(ex.Message);
            }
        }
```

```
        lblStatus.Text = "Done.";
        Application.DoEvents(  );

    }

    /// <summary>
    ///    Если нажата кнопка Cancel, закончить программу.
    /// </summary>
    private void btnCancel_Click(object sender, System.EventArgs e)
    {
        Application.Exit(  );
    }

    /// <summary>
    ///    Приказать корню снять выделение со всех узлов
    ///    нижних уровней.
    /// </summary>
    private void btnClear_Click(object sender, System.EventArgs e)
    {
        // Получить самый верхний узел каждого диска
        // и рекурсивно очистить его.
        foreach (TreeNode node in tvwSource.Nodes)
        {
            SetCheck(node, false);
        }
    }

    /// <summary>
    ///    Запросить подтверждение на удаление.
    ///    Составить список и удалить все файлы по очереди.
    /// </summary>
    private void btnDelete_Click(object sender, System.EventArgs e)
    {
        // Запросить подтверждение
        System.Windows.Forms.DialogResult result =
            MessageBox.Show(
            "Are you quite sure?",             // сообщение
            "Delete Files",                    // заголовок
            MessageBoxButtons.OKCancel,        // кнопки
            MessageBoxIcon.Exclamation,        // значок
            MessageBoxDefaultButton.Button2);  // кнопка по умолчанию
        // Если подтверждение получено,
        if (result == System.Windows.Forms.DialogResult.OK)
        {
            // Просмотреть в цикле список и удалить файлы.
            // Получить список выделенных файлов.
            ArrayList fileNames = GetFileList(  );

            foreach (FileInfo file in fileNames)
            {
                try
                {
                    // Вывести информацию в строке.
```

```
                    lblStatus.Text = "Deleting " +
                        txtTargetDir.Text + "\\" +
                        file.Name + "...";
                    Application.DoEvents(  );

                    // Опасная операция!
                    file.Delete( );
                }

                catch (Exception ex)
                {
                // Возможно, читатель захочет реализовать более сложную
                // обработку исключения, чем вывод сообщения на экран
                    MessageBox.Show(ex.Message);
                }
            }
            lblStatus.Text = "Done.";
            Application.DoEvents(  );
        }

    }

    /// <summary>
    ///    Получить полный путь к выбранному каталогу
    ///    и скопировать каталог в txtTargetDir.
    /// </summary>
    private void tvwTargetDir_AfterSelect(
        object sender,
        System.Windows.Forms.TreeViewEventArgs e)
    {
        // Получить полный путь к выбранному каталогу.
        string theFullPath = GetParentString(e.Node);

        // Если это не лист дерева, путь заканчивается
        // обратным слэшем.
        // Удалить обратный слэш
        if (theFullPath.EndsWith("\\"))
        {
            theFullPath = theFullPath.Substring(0,theFullPath.Length-1);
        }
        // Вывести путь в текстовое поле.
        txtTargetDir.Text = theFullPath;
    }

    /// <summary>
    ///    Установить или сбросить флажок каждого узла,
    ///    лежащего под данным, в соответствии
    ///    с текущим значением свойства Checked.
    /// </summary>
    private void tvwSource_AfterCheck(object sender,
        System.Windows.Forms.TreeViewEventArgs e)
    {
        // Вызов рекурсивного метода.
        // e.node - это узел, отмеченный пользователем.
        // Состояние флажка уже изменилось к данному моменту.
```

```
        // Поэтому передается значение e.node.Checked.
        SetCheck(e.Node,e.Node.Checked);
    }

    /// <summary>
    ///    Рекурсивно установить или сбросить флажки узлов.
    /// </summary>
    private void SetCheck(TreeNode node, bool check)
    {
        // Установить или сбросить флажок узла.
        foreach (TreeNode n in node.Nodes)
        {
            n.Checked = check;   // пометить узел

            // Рекурсивный вызов, если это ветвь дерева.
            if (n.Nodes.Count != 0)
            {
                SetCheck(n,check);
            }
        }
    }

    /// <summary>
    ///    Имея узел и массив-список ArrayList,
    ///    заполнить список именами выделенных файлов
    /// </summary>
    // Заполнить ArrayList полными путями
    // отмеченных файлов
    private void GetCheckedFiles(TreeNode node, ArrayList fileNames)
    {
        // Если это лист дерева…
        if (node.Nodes.Count == 0)
        {
            // и если узел был отмечен,
            if (node.Checked)
            {
                // получить полный путь и добавить его в arrayList.
                string fullPath = GetParentString(node);
                fileNames.Add(fullPath);
            }
        }
        else  // Если это ветвь дерева…
        {
            foreach (TreeNode n in node.Nodes)
            {
                GetCheckedFiles(n,fileNames);
            }
        }
    }

    /// <summary>
    ///    По данному узлу получить полный путь.
    /// </summary>
```

```
private string GetParentString(TreeNode node)
{
    // Если это корневой узел (c:\), возвратить текст.
    if(node.Parent == null)
    {
        return node.Text;
    }
    else
    {
        // Выполнить рекурсию вверх по дереву и получить путь;
        // затем добавить к нему данный узел и обратный слэш.
        // Если узел является листом, то слэш не добавлять
        return GetParentString(node.Parent) + node.Text +
            (node.Nodes.Count == 0  ? "" : "\\");
    }
}

/// <summary>
///    Метод, совместно используемый при удалении и копировании,
///    создает упорядоченный список всех выделенных файлов.
/// </summary>
private ArrayList GetFileList(  )
{
    // Создать несортированный список полных имен файлов
    ArrayList fileNames = new ArrayList(  );

    // для каждого копируемого файла добавить в ArrayList его
    // имя файла с полным путем.
    foreach (TreeNode theNode in tvwSource.Nodes)
    {
        GetCheckedFiles(theNode, fileNames);
    }
    // Создать список для хранения объектов fileInfo.
    ArrayList fileList = new ArrayList(  );

    // Каждое имя, хранящееся в списке, добавить в список
    // файлов, если оно соответствует файлу (а не каталогу).
    foreach (string fileName in fileNames)
    {
        // Создать файл с указанным именем,
        FileInfo file = new FileInfo(fileName);

        // проверить, существует ли он на диске.
        // Если это был каталог, результат проверки отрицательный.
        if (file.Exists)
        {

            fileList.Add(file);
        }
    }

    // Создать экземпляр интерфейса IComparer,
    IComparer comparer = (IComparer) new FileComparer(  );

    // передать объект comparer методу sort(), чтобы список
```

```
            // был отсортирован методом Compare() этого объекта.
            fileList.Sort(comparer);
            return fileList;
        }
    }
}
```

Документирующие комментарии XML

C# поддерживает особый стиль комментариев – *документирующие комментарии*, обозначаемые тремя слэшами (///). В примере 13.3 их довольно много. Текстовый редактор среды Visual Studio .NET распознает эти комментарии и помогает их отформатировать.

Компилятор C# транслирует эти комментарии в отдельный XML-файл. Создать такой файл можно вручную с помощью переключателя /doc в командной строке. Пример 13.3 можно скомпилировать, введя следующую строку:

```
csc Form1.cs /doc:XMLDoc.XML
```

Чтобы выполнить ту же операцию в Visual Studio .NET, щелкните правой кнопкой мыши по значку проекта в окне Solution Explorer или Class View и выберите в появившемся контекстном меню команду View → Property Pages. А затем раскройте папку Configuration Properties в появившемся диалоговом окне Property Pages. В этой папке выделите страницу Build и введите имя для свойства XML Documentation File, то есть имя XML-файла, который должен быть создан компилятором.

При любом из двух подходов будет создан файл XMLDoc.XML с комментариями в формате XML. В примере 13.4 приведен отрывок файла, который будет создан для приложения FileCopier, написанного в предыдущем разделе.

Пример 13.4. XML-файл для приложения FileCopier (фрагмент)

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>FileCopier</name>
    </assembly>
    <members>
        <member name="T:FileCopier.Form1">
            <summary>
             Форма, демонстрирующая реализацию Windows Forms
            </summary>
        </member>
        <member name="F:FileCopier.Form1.components">
            <summary>
            Required designer variable.
            </summary>
```

```
</member>
<member name="F:FileCopier.Form1.tvwTargetDir">
    <summary>
      Отображает потенциальный каталог назначения в виде дерева
    </summary>
</member>
<member name="F:FileCopier.Form1.tvwSource">
    <summary>
       Отображает исходный каталог в виде
       дерева с флажками для выделения нужных
       файлов или каталогов.
    </summary>
</member>
<member name="F:FileCopier.Form1.txtTargetDir">
```

Файл получается довольно длинным, и читать его в таком формате неудобно, хотя и возможно. Однако XML-код можно преобразовать в HTML, написав XSLT-файл. Кроме того, XML-документ можно сохранить в специальной базе данных для документации.

Самое простое, что можно сделать с документирующими комментариями, – позволить среде Visual Studio .NET сгенерировать отчет Code Comment Web Report. Для этого следует выбрать команду меню Tools → Build Comment Web Pages, а интегрированная среда разработки делает все остальное. Результатом будет набор HTML-файлов, которые можно просмотреть в Visual Studio .NET или с помощью броузера, как показано на рис. 13.9.

Каждый элемент класса, снабженный документирующим комментарием, включен в XML-файл с помощью тега `<member>`. Этот тег добавляется компилятором и содержит атрибут `name`, идентифицирующий элемент класса. Для «обогащения» создаваемого документа можно воспользоваться в комментарии стандартными тегами. Например, `<see>` позволяет сослаться на другой элемент класса, а `<exception>` документирует классы исключений. Подробное обсуждение документирующих комментариев XML выходит за рамки этой книги. Полный список тегов можно найти в документации, входящей в комплект Visual Studio .NET.

Развертывание приложения

Теперь, когда приложение работает, встает вопрос о его развертывании. Читателя, конечно, обрадует известие о том, что в .NET не используется системный реестр, причиняющий столько беспокойства. Достаточно просто скопировать сборку на другой компьютер.

Например, можно скомпилировать пример 13.3 в сборку по имени `FileCopier.exe`. Теперь скопируйте этот файл на другой компьютер и дважды щелкните по его имени. Все работает! Ни забот, ни хлопот.

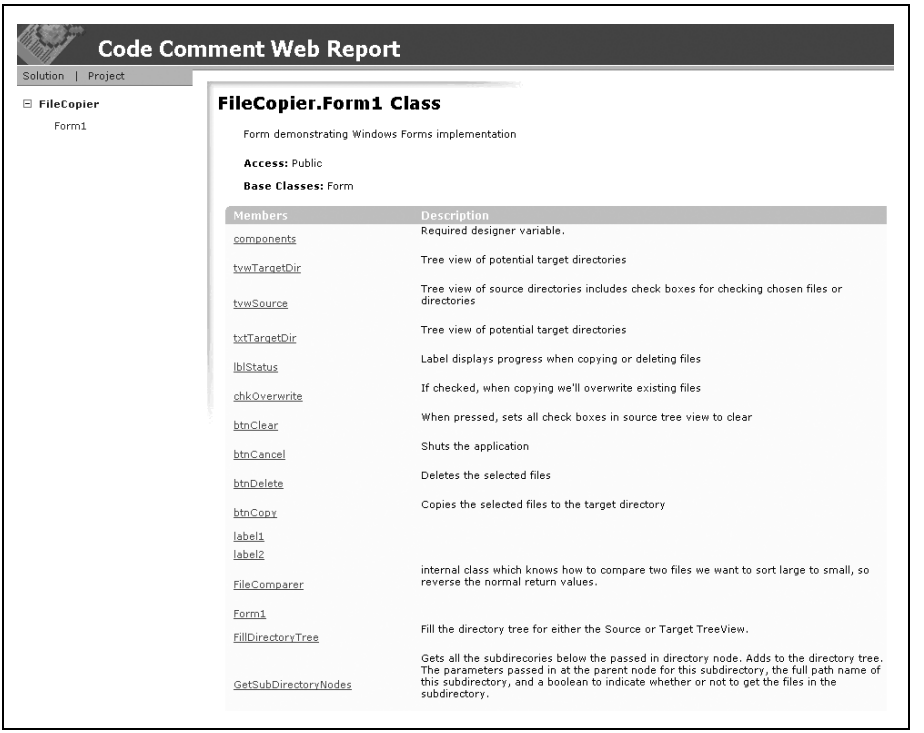

Рис. 13.9. *Отчет Code Comment Web Report*

Проекты развертывания

Для больших коммерческих проектов этого может оказаться недостаточно, каким бы заманчивым ни казался подобный подход. Пользователи предпочитают, чтобы при своей установке программа размещала файлы по каталогам, создавала значки и т. д.

Visual Studio .NET предоставляет программисту помощь в развертывании проекта. Чтобы воспользоваться ею, добавьте к проекту приложения специальный проект для его настройки при развертывании. Например, не выходя из проекта `FileCopier`, выберите команду меню File → Add Project → New Project и раскройте папку Setup and Deployment Projects в появившемся окне New Project. Это диалоговое окно примет вид, показанный на рис. 13.10.

Здесь программисту предлагается выбрать формат проекта. В случае проекта Windows, такого как `FileCopier`, можно выбрать:

Cab Project

Несколько маленьких файлов собираются в один удобный (и легко транспортируемый) пакет, аналогичный ZIP-файлу. Этот параметр может быть скомбинирован с другими.

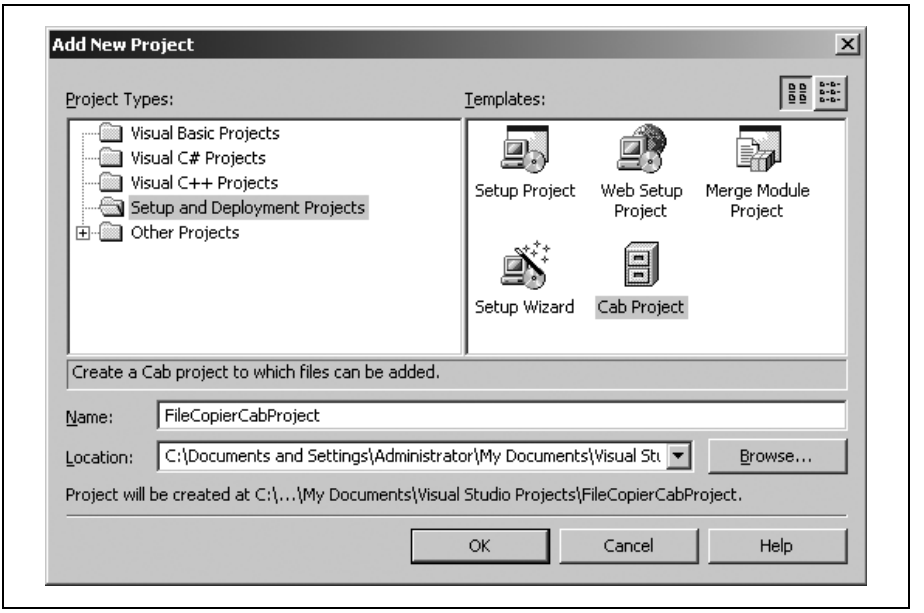

Рис. 13.10. *Диалоговое окно New Project*

Merge Module

При наличии нескольких проектов, имеющих общие файлы, этот параметр позволяет создавать промежуточные объединяющие модули, которые впоследствии могут быть интегрированы в другие проекты развертывания.

Setup Project

Эта опция создает файл, автоматически выполняющий установку файлов и ресурсов проекта.

Setup Wizard

Мастер, облегчающий создание одного из других типов проекта.

Remote Deploy Wizard

Мастер, создающий проект-инсталлятор, который может быть развернут автоматически.

Web Setup Project

Позволяет развернуть веб-проект.

Когда проект включает в себя много маленьких вспомогательных файлов, которые должны распространяться вместе с приложением (например, HTML-файлы, GIF-файлы и другие ресурсы), лучше всего выбрать Cab Project.

Чтобы посмотреть, как работает такой проект, выберите в меню File → Add Project → New Project. В окне New Project раскройте папку Setup and Dep-

loyment Projects, а затем выделите значок Cab Project. Дав проекту имя (например, FileCopierCabProject), нажмите кнопку OK. Проект будет добавлен в Solution Explorer, как показано на рис. 13.11.

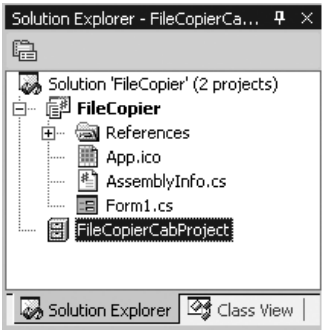

Рис. 13.11. *Проект Cab Project добавлен в группу*

Если щелкнуть по значку проекта правой кнопкой мыши, на экране появится контекстное меню. Выберите в нем команду Add, и появятся две новые команды: Project Output и File. Вторая позволяет добавить к Cab-проекту *совершенно* произвольный файл, а первая выводит на экран диалоговое окно, изображенное на рис. 13.12.

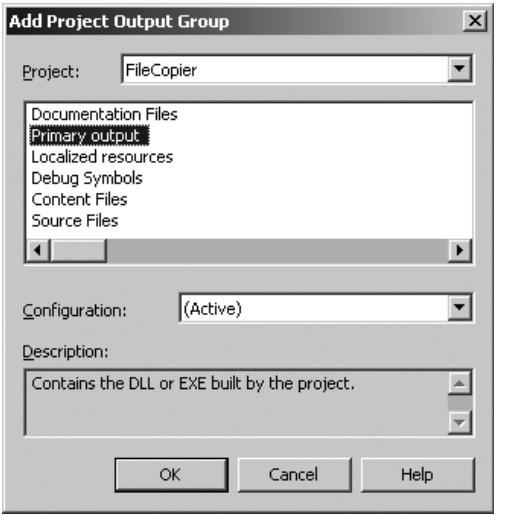

Рис. 13.12. *Диалоговое окно для опции Project Output*

Здесь можно выбирать группы файлов и добавлять их в коллекцию Cab. Группа Primary Output содержит распространяемую сборку выбранного проекта. Другие группы соответствуют необязательным файлам проекта, которые распространяются по желанию разработчика.

Для рассматриваемого примера, FileCopier, выделим строку Primary Output в списке диалогового окна Add Project Output Group. Результат этого выделения будет отражен в окне Solution Explorer, как это показано на рис. 13.13.

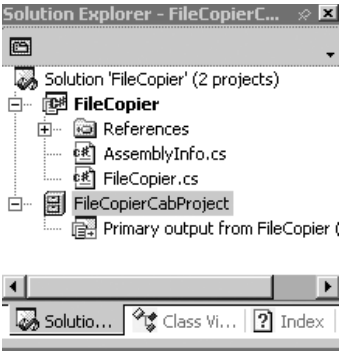

Рис. 13.13. *Модифицированный проект*

Теперь проект можно скомпилировать, и результатом будет CAB-файл (расположение которого отображается в окне Output). Проверить этот файл можно с помощью WinZip, как показано на рис. 13.14. Если у вас нет этой программы, можно воспользоваться утилитой *expand* (параметр -D перечисляет содержимое CAB-файла):

```
C:\...\FileCopierCabProject\Debug>expand -D FileCopierCabProject.CAB
Microsoft (R) File Expansion Utility  Version 5.1.2600.0
Copyright (C) Microsoft Corp 1990-1999.  All rights reserved.
filecopiercabproject.cab: OSDBF.OSD
filecopiercabproject.cab: FileCopier.exe
2 files total.
```

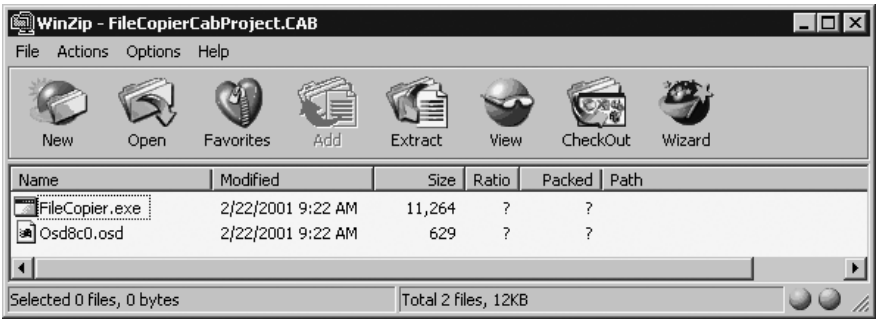

Рис. 13.14. *Содержимое CAB-файла*

Помимо ожидаемого EXE-файла файл сборки содержит второй, *Osd8c0.osd* (имя файла может быть и другим). Если его открыть, ока-

жется, что он представляет собой описание CAB-файла в формате XML, приведенное в примере 13.5.

Пример 13.5. Файл с описанием CAB-файла

```
<?XML version="1.0" ENCODING='UTF-8'?>
<!DOCTYPE SOFTPKG SYSTEM
"http://www.microsoft.com/standards/osd/osd.dtd">
<?XML::namespace href="http://www.microsoft.com/standards/osd/msicd.dtd"
as="MSICD"?>
<SOFTPKG NAME="FileCopierCabProject" VERSION="1,0,0,0">
        <TITLE> FileCopierCabProject </TITLE>
            <MSICD::NATIVECODE>
                <CODE NAME="FileCopier">
                    <IMPLEMENTATION>
                        <CODEBASE FILENAME="FileCopier.exe">
                        </CODEBASE>
                    </IMPLEMENTATION>
                </CODE>
            </MSICD::NATIVECODE>
</SOFTPKG>
```

Проект установки

Чтобы создать пакет для установки, следует в диалоговом окне New Project выделить значок Setup Project. Этот тип проекта очень гибок; он позволяет собрать все параметры настройки в одном установочном MSI-файле.

Если щелкнуть по значку проекта правой кнопкой мыши и выбрать команду Add в контекстном меню, то появится меню с дополнительными параметрами. Помимо Project Output и File оно будет содержать Merge Module и Component. Так же как и в случае с CAB-файлом, воспользуйтесь командой Add, чтобы добавить Primary Output в проект установки.

Объединяющие модули (Merge modules) – это общие фрагменты программ, которые впоследствии могут быть добавлены в законченный проект настройки. Опция Component позволяет добавлять в проект компоненты платформы .NET, которые понадобятся проекту, но могут отсутствовать на компьютере пользователя.

Пользовательский интерфейс для выполнения установки представляет собой диалоговое окно, разделенное надвое, чье содержимое определяется с помощью меню View. Чтобы вывести это меню на экране, достаточно щелкнуть по значку проекта правой кнопкой мыши (рис. 13.15).

Когда разработчик выбирает пункты меню View, окно среды Visual Studio .NET изменяется, отражая сделанный выбор и предоставляя соответствующие опции.

Например, если выбрать команду File System, откроется окно, разделенное пополам и содержащее в левой панели иерархический список и подробную информацию в правой панели. Если раскрыть папку Appli-

cation Folder, то можно увидеть множество файлов, уже добавленных в проект (Primary Output), что показано на рис. 13.16.

Разработчик имеет полное право удалять и добавлять файлы. Если щелкнуть правой кнопкой мыши по правой части окна, появится контекстное меню, изображенное на рис. 13.17.

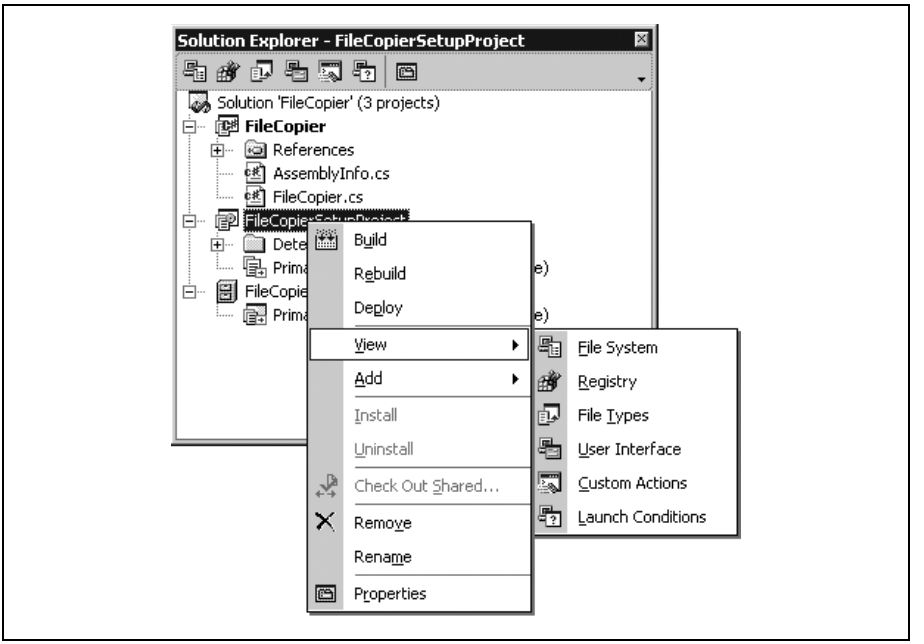

Рис. 13.15. *Меню View*

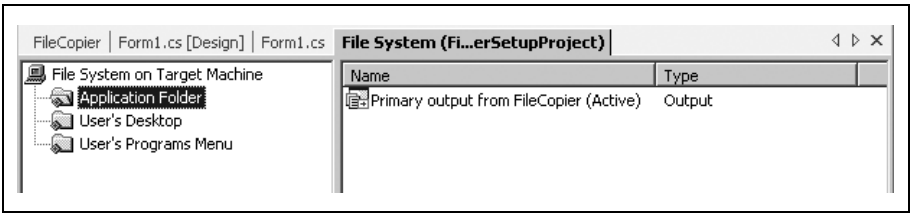

Рис. 13.16. *Папка Application Folder*

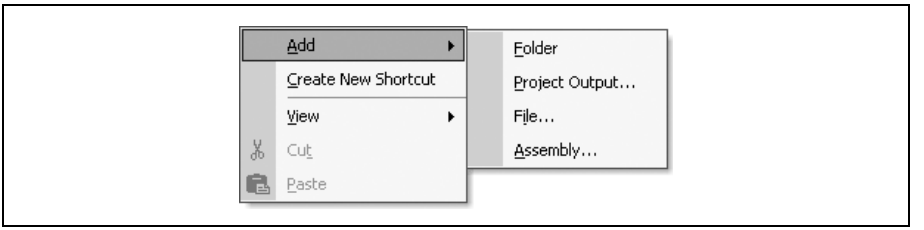

Рис. 13.17. *Контекстное меню правой части окна File System*

Таким образом, разработчик получает возможность добавлять в проект именно те файлы, которые он считает нужным.

Местоположение файлов

Папка, в которую будут помещены файлы проекта (папка Application Folder), задается на этапе создания проекта. Ее можно найти и изменить в окне Properties папки Application Folder. По умолчанию свойство DefaultLocation данного окна имеет значение: *[ProgramFilesFolder]\[Manufacturer]\[Product Name]*.

Здесь *ProgramFilesFolder* обозначает папку с файлами программы на компьютере пользователя.

Manufacturer (Производитель) и *Product Name* (Название продукта) являются свойствами проекта. Если щелкнуть по значку проекта установки и изучить его свойства, окажется, что среда Visual Studio .NET сделала некоторые предположения относительно значений этих свойств (рис. 13.18).

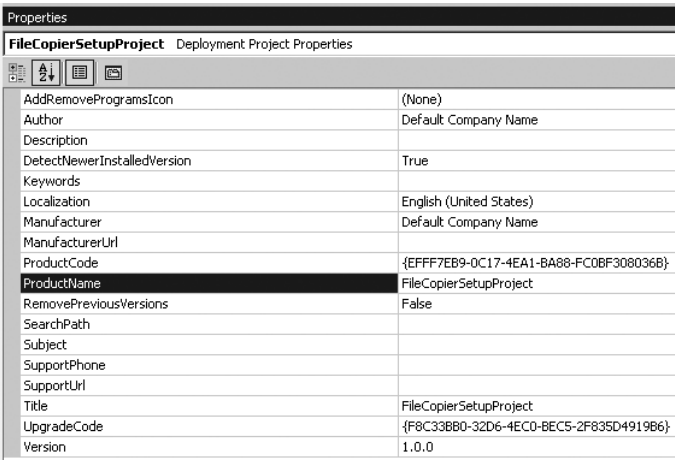

Рис. 13.18. *Свойства проекта настройки*

Любое из этих свойств может быть изменено. Например, можно изменить Manufacturer, и тогда проект будет сохранен в другом подкаталоге каталога Program Files.

Создание ярлыка

Если разработчик хочет, чтобы программа установки создала ярлык на рабочем столе пользователя, он должен щелкнуть правой кнопкой мыши по имени Primary Output в папке Application Folder. После этого можно создать ярлык и перетащить его в папку Desktop, показанную в окне File System (рис. 13.19).

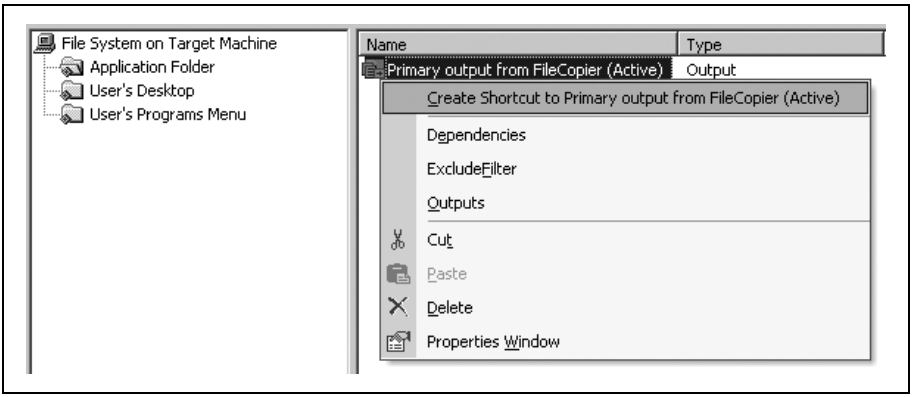

Рис. 13.19. *Создание ярлыка на рабочем столе пользователя*

Записи в папке My Documents

Разработчик может добавить записи в папку My Documents (Мои документы) на компьютере пользователя. Для этого сначала щелкните правой кнопкой мыши по папке File System on Target Machine, затем выберите Add Special Folder → User's Personal Data Folder. Теперь можно разместить необходимые файлы в созданной папке User's Personal Data Folder.

Ярлыки в меню Start

Помимо создания значка на рабочем столе пользователя можно создать папку в меню Start (Пуск) → Programs (Программы). Для этого нужно раскрыть папку User's Program Menu в левой части окна File System. Затем следует щелкнуть правой кнопкой мыши в правой части окна и выбрать команду Add → Folder в появившемся контекстном меню. После этого можно добавить Primary Output, либо перетащив его, либо щелкнув правой кнопкой мыши и выбрав пункт Add.

Добавление специальных папок

Кроме четырех основных папок, предоставляемых разработчику средой Visual Studio .NET (Application Folder, User's Desktop, User's Personal Data Folder, User's Program Menu), существует целый ряд дополнительных возможностей. Если щелкнуть правой кнопкой мыши по папке File System On Target Machine в окне File System, то появится меню, изображенное на рис. 13.20.

Это меню позволяет добавлять папки со шрифтами, добавлять записи в папку Favorites (Избранное) на компьютере пользователя и т. д. Назначение большинства пунктов меню понятно из их названий.

Другие окна меню View

До сих пор речь шла только об окне File System, вызванном из меню View (и изображенном на рис. 13.15).

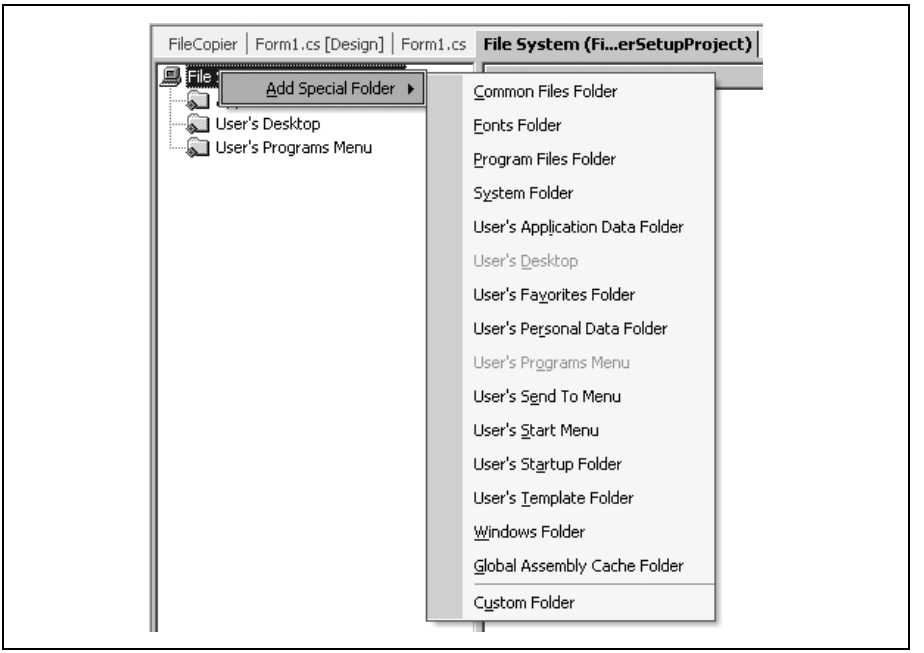

Рис. 13.20. *Меню с папками пользователя*

Внесение изменений в системный реестр

Окно Registry (щелкните правой кнопкой мыши по FileCopierSetupProject и выберите Registry из меню View) позволяет программисту сообщить проекту настройки, что необходимо внести изменение в файлы системного реестра на компьютере пользователя (рис. 13.21). Щелкните по любой папке в этом списке, чтобы отредактировать соответствующие свойства в окне Properties.

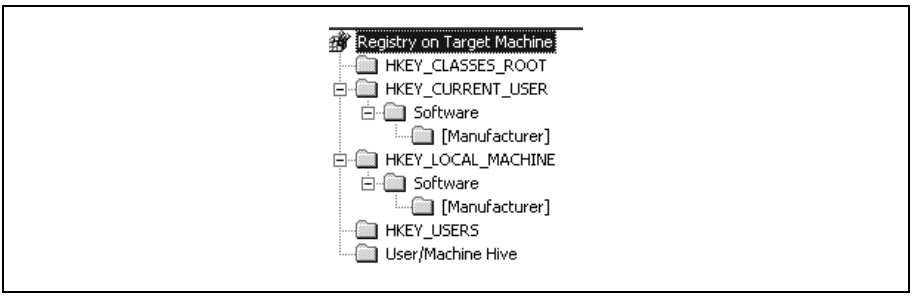

Рис. 13.21. *Настройка реестра*

Будьте осторожны! Нет ничего опаснее, чем внесение изменений в реестр. Для большинства приложений .NET это не требуется, поскольку программы, работающие на этой платформе, к реестру не обращаются.

Регистрация типов файлов

Пункт File Types в меню View позволяет программисту зарегистрировать на компьютере пользователя типы файлов, специфичные для данного проекта. Одновременно можно указать действия, предпринимаемые для файлов данного типа.

Управление пользовательским интерфейсом в процессе настройки

Пункт User Interface меню View позволяет разработчику определять внешний вид интерфейса на каждом шаге настройки. Сам процесс настройки представлен в виде иерархической структуры, изображенной на рис. 13.22.

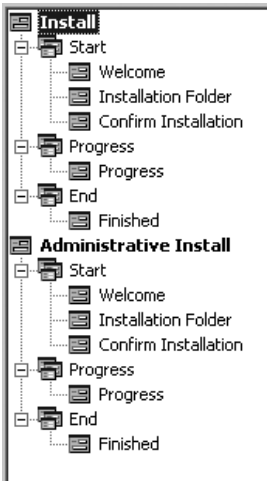

Рис. 13.22. *Процесс настройки*

Если щелкнуть по какому-либо значку процесса настройки, появятся свойства соответствующей формы. Например, если щелкнуть по узлу Welcome, расположенному под узлами Install и Start, появятся свойства, изображенные на рис. 13.23.

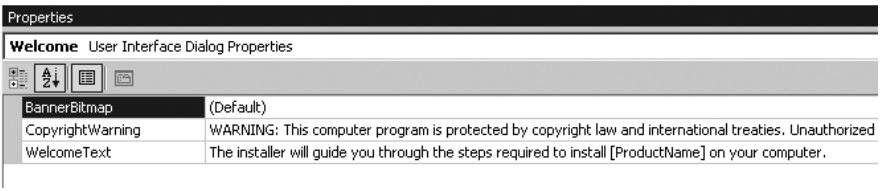

Рис. 13.23. *Форма Welcome*

Раздел свойств позволяет изменить заголовок (BannerBitmap) и текст приветствия (WelcomeText), появляющиеся в первом диалоговом окне. В последовательность диалоговых окон можно вставить стандартные окна, предоставляемые фирмой Microsoft, или диалоговые окна, определяемые пользователем.

Другие команды меню View

Если возможности, предоставляемые в стандартном процессе настройки, не удовлетворяют разработчика, он может выбрать команду Custom Actions в меню View. Кроме того, разработчик может определить условия запуска (пункт Launch conditions) для самого процесса настройки.

Компиляция проекта настройки

Выбрав нужные команды меню и установив требуемые параметры, выберите Configuration Manager из меню Build и убедитесь, что проект настройки (Setup Project) включен в текущую конфигурацию. Теперь программист может скомпилировать его. Результатом будет один установочный файл (*FileCopierSetupProject.msi*), который можно распространять среди пользователей.

14

Доступ к данным с помощью ADO.NET

Многим коммерческим приложениям необходимо взаимодействовать с базами данных. Платформа .NET Framework предоставляет программисту богатый выбор объектов, позволяющих обращаться к базам данных. Совокупность классов этих объектов носит имя ADO.NET.

ADO.NET очень напоминает своего предшественника, ADO. Главное различие между ними заключается в том, что ADO.NET представляет собой *отсоединенную* (*disconnected*) архитектуру данных. При отсоединенной архитектуре данные извлекаются из базы и кэшируются на локальном компьютере. Там приложение обрабатывает их и связывается с базой данных только тогда, когда возникает необходимость изменить в ней какие-либо записи или получить новые данные.

В отсоединении от базы данных есть много преимуществ. Самое важное из них состоит в том, что такой подход позволяет избежать многих проблем, возникающих при работе с подсоединенными объектами данных, которые плохо масштабируются. Подключение к базе данных активно использует ресурсы, и одновременная поддержка тысяч (или десятков тысяч) подключений представляет собой трудную задачу. Отсоединенная архитектура использует ресурсы не так интенсивно.

Первый раз ADO.NET связывается с базой данных для получения информации, а затем связывается снова – уже для внесения изменений, сделанных вами. Большинство приложений значительную часть времени просто читают данные и выводят их на экран; ADO.NET предоставляет приложению отсоединенное подмножество данных, которые можно читать и выводить.

Отсоединенные объекты данных работают в режиме, напоминающем веб-сеансы. Все веб-сеансы являются обособленными, и их состояния не сохраняются между запросами веб-страниц. Отсоединенная архи-

тектура данных позволяет осуществлять более «чистое» взаимодействие с базами данных в Сети.

Реляционные базы данных и язык SQL

Хотя базам данных можно посвятить целую книгу, а еще одну – языку SQL, овладеть основами этих технологий ничуть не трудно. *База данных* – это хранилище информации. *Реляционная база данных* организует эту информацию в таблицы. Примерами могут служить база данных Northwind, поставляемая с Microsoft SQL Server 7, SQL Server 2000, а также со всеми версиями Microsoft Access.

Таблицы, записи и столбцы

База данных Northwind описывает воображаемую фирму, перепродающую пищевые продукты. Данные в Northwind разделяются на 13 таблиц, в том числе Customers (Заказчики), Employees (Сотрудники), Orders (Заказы), Order Details (Подробности о заказах), Products (Продукты) и т. д.

Таблица реляционной базы данных состоит из строк, каждая из которых представляет одну запись. Строки содержат поля, соответствующие столбцам таблицы, причем строки одной таблицы имеют одинаковую структуру полей. Например, таблица Orders имеет столбцы `OrderID`, `CustomerID`, `EmployeeID`, `OrderDate` и т. д.

По каждому заказу необходима такая информация, как название фирмы-заказчика, ее адрес, имя сотрудника для контакта и т. д. Подобную информацию можно хранить вместе с заказом, но это крайне неэкономно. Куда разумнее иметь отдельную таблицу, каждая строка которой соответствует одному заказчику. В таблице Customers есть столбец `CustomerID`. Каждому заказчику присваивается уникальный идентификационный номер, и это поле объявляется *первичным ключом* таблицы. *Первичный ключ* (*primary key*) – это одно или несколько полей, идентифицирующих запись в таблице уникальным образом.

В таблице Orders тоже есть столбец `CustomerID`, но для нее это *внешний ключ* (*foreign key*). Внешний ключ – это столбец (или комбинация нескольких столбцов), являющийся первичным (иными словами, уникальным) ключом другой таблицы. В таблице Orders столбец `CustomerID` (который является первичным ключом таблицы Customers) используется для идентификации заказчика, разместившего заказ. Чтобы определить, например, адрес, по которому следует отсылать заказ, можно по ключу `CustomerID` найти запись о заказчике в таблице Customers.

Особенно полезны внешние ключи, когда между таблицами установлены отношения «один-ко-многим» и «многие-к-одному». Разделяя информацию между таблицами, связанными внешними ключами, разработчик избегает дублирования информации в записях. Напри-

мер, один заказчик может сделать несколько заказов, поэтому хранить в каждой записи одну и ту же информацию о заказчике (имя, телефон, кредитный лимит и т. д.), по меньшей мере, неэкономно. Процесс выделения избыточной информации в отдельную таблицу называется *нормализацией*.

Нормализация

Нормализация не просто делает базы данных эффективнее – она уменьшает вероятность порчи данных. Если хранить имя заказчика в обеих таблицах, Customers и Orders, всегда есть риск, что изменения в одной таблице не будут отражены в другой. Если изменить адрес заказчика в таблице Customers, нет гарантии, что он будет изменен в каждой записи таблицы Orders (для проверки этого потребуется огромная работа). А если в таблице Orders хранится лишь идентификатор заказчика (`CustomerID`), то можно спокойно менять адрес в таблице Customers, и изменение будет автоматически отражено в каждом заказе.

Подобно тому как программисты, пишущие на C#, предпочитают, чтобы ошибки выявлялись на этапе компиляции, а не выполнения, разработчики баз данных рассчитывают на помощь самих баз данных в борьбе за сохранность информации. Компилятор C# способствует созданию безошибочных программ, поскольку требует строгого соблюдения синтаксиса языка; например, нельзя пользоваться переменной до ее определения. Аналогичным образом SQL Server и другие современные реляционные базы данных помогают избежать ошибок, строго соблюдая ограничения, накладываемые разработчиком. Например, в таблице Customers столбец `CustomerID` помечен как первичный ключ. Тем самым в базе данных создается ограничение, гарантирующее уникальность каждого значения ключа. Если ввести запись о фирме-заказчике с названием Liberty Associates, Inc. с идентификатором LIBE, а потом попытаться добавить фирму Liberty Mutual Funds с таким же идентификатором, то база данных отклонит вторую запись на основании ограничения первичного ключа.

Декларативная целостность ссылок

В реляционных базах данных *декларативная целостность ссылок* (*Declarative Referential Integrity*) используется для наложения ограничений на отношения между различными таблицами. Например, на таблицу Orders может быть наложено ограничение, требующее, чтобы идентификатор заказчика (`CustomerID`) в заказе обязательно представлял запись в таблице Customers. Такое требование позволяет исключить ошибки двух типов. Во-первых, не удается ввести в таблицу Orders запись с недопустимым идентификатором заказчика. Во-вторых, не удается удалить запись из таблицы Customers, если ее `CustomerID` используется хотя бы в одном заказе. Так осуществляется защита целостности данных и отношений между ними.

Язык SQL

Самым популярным языком запросов к базам данных и обработки данных является SQL. SQL – декларативный (непроцедурный) язык, и программисту, работающему например на C#, требуется какое-то время, чтобы привыкнуть к нему.

Важнейшим элементом языка SQL является *запрос*. Запрос – это команда, которая возвращает набор записей из базы данных.

Пусть, например, требуется просмотреть все названия фирм (столбец `CompanyName`) и идентификаторы заказчиков (столбец `CustomerID`) из таблицы Customers при условии, что фирма-заказчик находится в Лондоне. Для этого нужно написать следующий оператор:

```
Select CustomerID, CompanyName from Customers where city = 'London'
```

Он возвратит следующие записи:

```
CustomerID CompanyName
---------- ----------------------------------------
AROUT      Around the Horn
BSBEV      B's Beverages
CONSH      Consolidated Holdings
EASTC      Eastern Connection
NORTS      North/South
SEVES      Seven Seas Imports
```

Язык SQL способен формулировать гораздо более сложные запросы. Предположим, менеджер фирмы Northwind захочет узнать, какие продукты были закуплены в июле 1996 года заказчиком Vins et alcools Chevalier. Это будет далеко не тривиальный запрос. Таблица Order Details содержит идентификаторы `ProductID` для всех продуктов в любом заказе. Таблица Orders «знает», какие идентификаторы `CustomerID` связаны с заказами. В таблице Customers по идентификатору можно получить информацию о заказчике, а в таблице Products по идентификатору продукта можно узнать его наименование. Как связать все это воедино? С помощью следующего запроса:

```
select  o.OrderID, productName
from [Order Details] od
join orders o on o.OrderID = od.OrderID
join products p on p.ProductID = od.ProductID
join customers c on o.CustomerID = c.CustomerID
where c.CompanyName = 'Vins et alcools Chevalier'
and orderDate >= '7/1/1996' and orderDate <= '7/31/1996'
```

Здесь у базы данных запрашивается идентификатор заказа (`OrderID`) и название продукта (`ProductName`) из соответствующих таблиц. При этом вначале изучается таблица Order Details (для краткости названная `od`), затем результат объединяется с таблицей Orders для тех записей, у ко-

торых идентификатор заказа в таблице Order Details совпадает с идентификатором заказа в таблице Orders.

Когда объединяются две таблицы, возможны две формулировки этого процесса. Либо «выбрать каждую запись, имеющуюся в одной из таблиц» (это называется *внешним объединением*), либо, как в данном случае, «выбрать только те записи, которые имеются в обеих таблицах» (называется *внутренним объединением*). Иными словами, внутреннее объединение возвращает только те записи из таблицы Orders, у которых поле `OrderID` имеет то же значение, что и запись из таблицы Order Details (`on o.Orderid = od.Orderid`).

В языке SQL объединения являются внутренними по умолчанию. Конструкция `join orders` полностью эквивалентна конструкции `inner join orders`.

Далее в команде SQL выражена просьба к базе данных создать внутреннее объединение с таблицей Products, выбирая записи, у которых идентификатор продукта `ProductID` в таблице Products совпадает с идентификатором продукта в таблице Order Details.

После этого создается внутреннее объединение с таблицей Customers для тех строк, у которых идентификаторы `CustomerID` совпадают как в таблице Orders, так и в таблице Customers.

Наконец, выдается инструкция базе данных об ограничении поиска только теми строками, которые содержат нужное название фирмы и датированы июлем 1996 года.

Такой набор ограничений приводит к следующим результатам:

```
OrderID     ProductName
----------- ----------------------------------------
10248       Queso Cabrales
10248       Singaporean Hokkien Fried Mee
10248       Mozzarella di Giovanni
```

Из этого вывода видно, что в июле 1996 года заказчик, интересующий менеджера, сделал только один заказ (номер 10248). Этому заказу в таблице Order Details соответствуют три записи. По идентификаторам продуктов в этих записях удалось узнать названия продуктов из таблицы Products.

Средства языка SQL позволяют не только отыскивать и читать данные, но также создавать, обновлять и удалять таблицы и вообще управлять как содержимым, так и структурой базы данных.

Полное описание языка SQL и рекомендации по его применению содержатся в книге Клайна (Kline), Гоулда (Gould) и Заневского (Zanevsky) «Transact SQL Programming», выпущенной издательством O'Reilly & Associates в 1999 году.

Объектная модель ADO.NET

Объектная модель ADO.NET довольно велика, но по сути своей является просто набором классов. Самый важный из них – класс DataSet. Этот класс представляет подмножество базы данных, которое находится на локальном компьютере без непрерывной связи с базой данных.

Периодически объект DataSet восстанавливает соединение с родительской базой данных для внесения в нее изменений, сделанных в объекте DataSet, и для внесения в него изменений, произошедших в базе данных.

Такой подход весьма эффективен, но при этом от объекта DataSet требуется, чтобы он был устойчивым подмножеством базы данных и включал в себя не просто несколько строк какой-то таблицы, а целый набор таблиц со всеми метаданными, необходимыми для отражения отношений и ограничений, присутствующих в исходной базе данных. К счастью, ADO.NET удовлетворяет всем этим требованиям.

Объект DataSet состоит из объектов DataTable и DataRelation, доступных через его свойства. Свойство Tables возвращает коллекцию DataTableCollection, которая содержит все объекты DataTable.

Объекты DataTable и DataColumn

Объект DataTable может быть создан программно либо в результате запроса к базе данных. Объект DataTable обладает рядом открытых свойств, в число которых входит коллекция Columns, которая возвращает объект DataColumnCollection, состоящий из объектов DataColumn. Каждый объект DataColumn представляет собой столбец таблицы.

Объекты DataRelation

Кроме коллекции Tables объект DataSet имеет свойство Relations, которое возвращает коллекцию DataRelationCollection, состоящую из объектов DataRelation. Каждый такой объект представляет собой отношение между двумя таблицами, выраженное через объекты DataColumn. Например, в базе данных Northwind таблицы Customers и Orders имеют отношение, выраженное через столбец CustomerID.

Это отношение вида «один-ко-многим» или «предок-к-потомку». У каждого заказа есть единственный заказчик, но у заказчика может быть любое количество заказов.

Коллекция Rows

Коллекция Rows объекта DataTable возвращает набор строк указанной таблицы. Эта коллекция используется для изучения результатов запроса к базе данных. Циклический перебор элементов коллекции Rows позволяет проанализировать каждую запись. Программистов, имею-

щих опыт работы с моделью ADO, может удивить отсутствие объекта RecordSet и его команд moveNext и movePrevious. В ADO.NET программист не перебирает элементы DataSet; вместо этого он обращается к нужной таблице и перебирает элементы коллекции Rows (как правило, в цикле foreach). Такой подход будет продемонстрирован уже в первом примере этой главы.

Адаптер данных

Объект DataSet является абстрактным представлением реляционной базы данных. В модели ADO.NET «мостом» между объектом DataSet и источником данных (базой данных) служит DataAdapter. Он предоставляет программисту метод Fill(), позволяющий извлекать информацию из базы данных и заполнять ею объект DataSet.

Объекты DBCommand и DBConnection

Объект DBConnection представляет связь с источником данных. Эта связь может быть одновременно использована несколькими командными объектами. Объект DBCommand позволяет послать базе данных команду (как правило, команду SQL или хранимую процедуру). Нередко эти объекты создаются неявно в момент создания объекта DataSet, но обращаться к ним можно явно, что демонстрирует пример, приведенный чуть ниже.

Объект DataAdapter

Вместо жесткой привязки объекта DataSet к архитектуре базы данных модель ADO.NET предоставляет объект DataAdapter в качестве посредника между DataSet и базой. Разъединение объекта DataSet и базы данных позволяет одному объекту этого класса получать информацию из нескольких баз данных или иных источников информации.

Приступаем к работе с моделью ADO.NET

Достаточно теории! Напишем реальное приложение и посмотрим, как оно работает. Иногда работа с ADO.NET представляет определенные трудности, но для большинства запросов модель оказывается на удивление простой.

В примере, приведенном ниже, создается простая Windows-форма с единственным элементом – окном списка с именем lbCustomers. Этот список будет заполняться информацией из таблицы Customers базы данных Northwind.

Начнем с создания объекта DataAdapter:

```
SqlDataAdapter DataAdapter =
new SqlDataAdapter(commandString, connectionString);
```

Он имеет два аргумента, commandString и connectionString. Первый из них является командой языка SQL, которая считает данные в объект DataSet:

```
string commandString = "Select CompanyName, ContactName from Customers";
```

Аргумент connectionString — это строка, необходимая для установки связи с базой данных. На компьютере автора установлен SQL Server, причем именем системного администратора является sa, а пароль — пустая строка. (Автор знает, что это недопустимо, и дает честное слово все исправить, как только закончит книгу.)

```
string connectionString =
    "server=localhost; uid=sa; pwd=; database=northwind";
```

Если у вас не установлен SQL-сервер, выберите *Samples and Quickstart Tutorials* из группы программ *Microsoft .NET Framework SDK*. На экране появится веб-страница, позволяющая установить базу данных примеров (.NET Framework Samples Database), которая включает в себя установку SQL-сервера. Затем установите QuickStarts, при этом будет создан пример базы данных Northwind. Чтобы связаться с ним, потребуется такая строка:

```
"server=(local)\\NetSDK; Trusted_Connection=yes; database=northwind"
```

Имея объект DataAdapter, можно приступать к созданию объекта DataSet и заполнению его информацией, полученной с помощью SQL-команды select:

```
DataSet DataSet = new DataSet();
DataAdapter.Fill(DataSet,"Customers");
```

Вот и все. Объект DataSet готов. К нему можно обращаться с запросами и выполнять с его данными самые разные операции. Объект DataSet содержит коллекцию таблиц, но сейчас интерес представляет только первая, поскольку была прочитана только одна запись:

```
DataTable dataTable = DataSet.Tables[0];
```

Строки, полученные с помощью команды языка SQL, можно прочитать из таблицы и добавить в окно списка:

```
foreach (DataRow dataRow in dataTable.Rows)
{
    lbCustomers.Items.Add(
        dataRow["CompanyName"] + " (" + dataRow["ContactName"] + ")" );
}
```

В этом фрагменте программы список заполняется названиями фирм и именами сотрудников, ответственных за заказ продуктов. Эта информация берется из таблицы, полученной из базы данных по SQL-запросу. Полный исходный текст приведен в примере 14.1.

Пример 14.1. Работа с моделью ADO.NET

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ProgrammingCSharpWinForm
{

   public class ADOForm1 : System.Windows.Forms.Form
   {
      private System.ComponentModel.Container components;
      private System.Windows.Forms.ListBox lbCustomers;

      public ADOForm1(  )
      {
         InitializeComponent(  );

         // установить соединение с локальным сервером и
         // связь с базой данных northwind
         string connectionString = "server=(local)\\NetSDK;" +
         "Trusted_Connection=yes; database=northwind";

         // получить записи из таблицы Customers
         string commandString =
         "Select CompanyName, ContactName from Customers";

         // создать объекты команды и DataSet
         SqlDataAdapter DataAdapter =
         new SqlDataAdapter(
         commandString, connectionString);
         DataSet DataSet = new DataSet(  );

         // заполнить DataSet
         DataAdapter.Fill(DataSet,"Customers");

         // получить одну таблицу из DataSet
         DataTable dataTable = DataSet.Tables[0];

         // для каждой строки таблицы вывести данные
         foreach (DataRow dataRow in dataTable.Rows)
         {
            lbCustomers.Items.Add(
               dataRow["CompanyName"] +
               " (" + dataRow["ContactName"] + ")" );
         }

      }

      protected override void Dispose(bool disposing)
      {
         if (disposing)
```

```
            {
                if (components == null)
                {
                    components.Dispose(  );
                }
            }
            base.Dispose(disposing);
        }

        private void InitializeComponent(  )
        {
            this.components =
               new System.ComponentModel.Container (  );
            this.lbCustomers = new System.Windows.Forms.ListBox (  );
            lbCustomers.Location = new System.Drawing.Point (48, 24);
            lbCustomers.Size = new System.Drawing.Size (368, 160);
            lbCustomers.TabIndex = 0;
            this.Text = "ADOFrm1";
            this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
            this.ClientSize = new System.Drawing.Size (464, 273);
            this.Controls.Add (this.lbCustomers);
        }

        public static void Main(string[] args)
        {
            Application.Run(new ADOForm1(  ));
        }
    }
}
```

Нескольких строчек программы оказалось достаточно для извлечения набора записей из базы данных и вывода их содержимого в виде списка, изображенного на рис. 14.1.

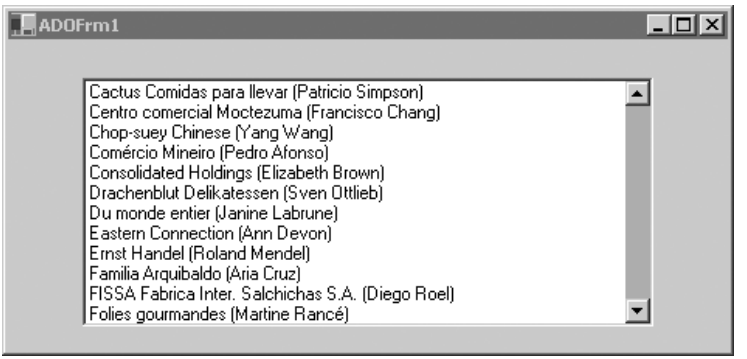

Рис. 14.1. *Результат работы программы из примера 14.1*

Восемь строк программы решают следующие задачи:

- Создание строки для установки соединения:

```
string connectionString = "server=(local)\\NetSDK;" +
"Trusted_Connection=yes; database=northwind";
```

- Создание строки с командой `select`:

```
string commandString = "Select CompanyName, ContactName from Customers";
```

- Создание объекта `DataAdapter` и передача ему строки соединения и строки с командой:

```
SqlDataAdapter DataAdapter =
new SqlDataAdapter(commandString, connectionString);
```

- Создание объекта `DataSet`:

```
DataSet DataSet = new DataSet();
```

- Заполнение объекта `DataSet` информацией из таблицы Customers (с помощью объекта `DataAdapter`):

```
DataAdapter.Fill(DataSet,"Customers");
```

- Извлечение объекта `DataTable` из объекта `DataSet`:

```
DataTable dataTable = DataSet.Tables[0];
```

- Заполнение списка информацией из объекта `DataTable`:

```
foreach (DataRow dataRow in dataTable.Rows)
{
   lbCustomers.Items.Add(
      dataRow["CompanyName"] +
      " (" + dataRow["ContactName"] + ")" );
}
```

Использование управляемых поставщиков OLE DB

В предыдущем примере был использован один из двух управляемых поставщиков, доступных в модели ADO.NET. Эти два поставщика – управляемый поставщик SQL Server и управляемый поставщик OLE DB. Первый оптимизирован для работы с SQL Server и может работать только с его базами данных. Управляемый поставщик OLE DB реализует более общий подход и может работать с любой базой данных OLE DB, включая Access.

Пример 14.1 можно переписать так, что он будет обращаться к базе данных Northwind с помощью Access, а не SQL Server, и изменения в программе будут весьма незначительны. Прежде всего потребуется другая строка связи:

```
string connectionString =
    "provider=Microsoft.JET.OLEDB.4.0; data source = c:\\nwind.mdb";
```

Здесь устанавливается связь с базой данных Northwind, расположенной на диске C. (Возможно, на компьютере читателя путь к базе данных будет другим.)

Теперь следует изменить тип объекта `DataAdapter` с `SqlDataAdapter` на `OleDbDataAdapter`:

```
OleDbDataAdapter DataAdapter =
    new OleDbDataAdapter (commandString, connectionString);
```

Кроме того, нужно не забыть оператор `using` для пространства имен `OleDb`:

```
using System.Data.OleDb;
```

Далее следует внести изменения, отражающие выбор другого управляемого поставщика. Для каждого класса, имя которого начинается с «Sql», существует соответствующий класс, имя которого начинается с «OleDb». Пример 14.2 содержит полную OLE DB-версию программы из примера 14.1.

Пример 14.2. Использование управляемого поставщика OLE DB

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.OleDb;

namespace ProgrammingCSharpWinForm
{

   public class ADOForm1 : System.Windows.Forms.Form
   {
      private System.ComponentModel.Container components;
      private System.Windows.Forms.ListBox lbCustomers;

      public ADOForm1(  )
      {
         InitializeComponent(  );

         // соединиться с базой данных Northwind
         string connectionString =
         "provider=Microsoft.JET.OLEDB.4.0;“ + “data source = c:\\nwind.mdb";


         // получить записи из таблицы Customers
         string commandString =
         "Select CompanyName, ContactName from Customers";

         // создать объект команды и DataSet
         OleDbDataAdapter DataAdapter =
           new OleDbDataAdapter(commandString, connectionString);

         DataSet DataSet = new DataSet(  );
```

```
            // заполнить DataSet
            DataAdapter.Fill(DataSet,"Customers");

            // получить одну таблицу из DataSet
            DataTable dataTable = DataSet.Tables[0];

            // для каждой строки таблицы вывести данные
            foreach (DataRow dataRow in dataTable.Rows)
            {
               lbCustomers.Items.Add(
                  dataRow["CompanyName"] +
                  " (" + dataRow["ContactName"] + ")" );
            }

         }

         protected override void Dispose(bool disposing)
         {
            if (disposing)
            {
                if (components == null)
                {
                    components.Dispose(  );
                }

            }
            base.Dispose(disposing);
         }

         private void InitializeComponent(  )
         {
            this.components =
               new System.ComponentModel.Container (  );
            this.lbCustomers = new System.Windows.Forms.ListBox (  );
            lbCustomers.Location = new System.Drawing.Point (48, 24);
            lbCustomers.Size = new System.Drawing.Size (368, 160);
            lbCustomers.TabIndex = 0;
            this.Text = "ADOFrm1";
            this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
            this.ClientSize = new System.Drawing.Size (464, 273);
            this.Controls.Add (this.lbCustomers);
         }

         public static void Main(string[] args)
         {
            Application.Run(new ADOForm1(  ));
         }
      }
   }
```

Результат работы кода примера 14.2 идентичен результату предыдущего примера 14.1 (рис. 14.2).

Управляемый поставщик OLE DB является обобщением управляемого поставщика SQL и может, в частности, применяться для связи с SQL

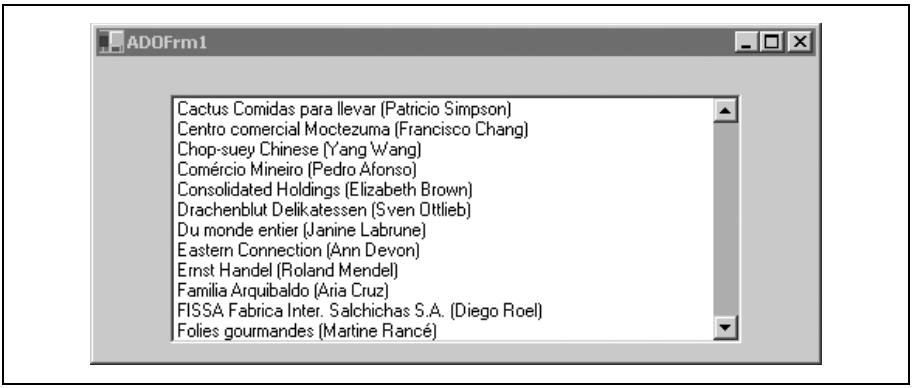

Рис. 14.2. *Использование управляемого поставщика OLE DB*

Server, как и с любым другим объектом OLE DB. Поскольку поставщик SQL оптимизирован для работы с SQL Server, то при работе с ним он будет эффективнее. Со временем появятся и другие специализированные управляемые поставщики.

Использование элементов управления с привязкой данных

Модель ADO.NET предоставляет хорошую поддержку объектам с привязкой данных. Так называются объекты, которые можно связать с конкретным набором данных, полученным от базы данных с помощью модели ADO.NET.

Простым примером элемента управления, связанного с данными, является элемент управления `DataGrid`, предоставляемый как средой Windows Forms, так и средой Web Forms.

Заполнение элемента управления DataGrid

В простейшем варианте реализовать элемент управления `DataGrid` очень легко. Как и прежде, вначале создадим объект `DataSet`, а затем заполним его информацией из таблицы Customers базы данных Northwind. Впрочем, на этот раз циклический просмотр строк с занесением информации в список выполняться не будет. Вместо этого таблица Customers будет просто привязана к элементу управления `DataGrid`.

В качестве иллюстрации возьмем пример 14.1, заменив в форме окно списка на элемент управления `DataGrid`. По умолчанию среда Visual Studio .NET даст этому элементу имя `DataGrid1`. Переименуем его в `CustomerDataGrid`. После того как набор записей будет создан и заполнен, объект `DataGrid` привязывается к нему через свойство `DataSource`:

```
CustomerDataGrid.DataSource = DataSet.Tables["Customers"].DefaultView;
```

Полный исходный текст этой программы содержится в примере 14.3.

Пример 14.3. Использование элемента управления DataGrid

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;

namespace ProgrammingCSharpWindows.Form
{

   public class ADOForm3 : System.Windows.Forms.Form
   {
      private System.ComponentModel.Container
         components;
      private System.Windows.Forms.DataGrid
         CustomerDataGrid;

      public ADOForm3(  )
      {
         InitializeComponent(  );

         // создать командную строку и строку соединения
         string connectionString = "server=(local)\\NetSDK;" +
         "Trusted_Connection=yes; database=northwind";
         string commandString =
            "Select CompanyName, ContactName, ContactTitle, "
            + "Phone, Fax from Customers";

         // создать DataSet и заполнить его
         SqlDataAdapter DataAdapter =
            new SqlDataAdapter(commandString, connectionString);
         DataSet DataSet = new DataSet(  );
         DataAdapter.Fill(DataSet,"Customers");

         // связать DataSet с DataGrid
         CustomerDataGrid.DataSource=
            DataSet.Tables["Customers"].DefaultView;
      }

      protected override void Dispose(bool disposing)
      {
         if (disposing)
         {
             if (components == null)
             {
                 components.Dispose(  );
             }
         }
         base.Dispose(disposing);
      }
```

```
        private void InitializeComponent(  )
        {
           this.components =
              new System.ComponentModel.Container (  );
           this.CustomerDataGrid =
              new System.Windows.Forms.DataGrid (  );
           CustomerDataGrid.BeginInit (  );
           CustomerDataGrid.Location =
              new System.Drawing.Point (8, 24);
           CustomerDataGrid.Size =
              new System.Drawing.Size (656, 224);
           CustomerDataGrid.DataMember = "";
           CustomerDataGrid.TabIndex = 0;
           CustomerDataGrid.Navigate +=
              new System.Windows.Forms.NavigateEventHandler
              (this.dataGrid1_Navigate);
           this.Text = "Using the Data Grid";
           this.AutoScaleBaseSize =
              new System.Drawing.Size (5, 13);

           this.ClientSize = new System.Drawing.Size (672, 273);
           this.Controls.Add (this.CustomerDataGrid);
           CustomerDataGrid.EndInit (  );
        }

        protected void dataGrid1_Navigate
           (object sender, System.Windows.Forms.NavigateEventArgs ne)
        {

        }

        public static void Main(string[] args)
        {
           Application.Run(new ADOForm3(  ));
        }
     }
  }
```

Программа оказалась на удивление простой, а результаты (показанные на рис. 14.3) впечатляют. Обратите внимание, что поля записей представлены столбцами элемента `DataGrid`, а имена полей стоят в заголовках столбцов. И все это является поведением объекта `DataGrid` по умолчанию.

Настройка объекта DataSet

У программиста есть возможность контролировать каждый аспект создания объекта `DataSet` и не ограничивать себя настройками по умолчанию! В предыдущих примерах при создании объекта `DataSet` ему передавались аргументы `commandString` и `connectionString`:

```
SqlDataAdapter DataAdapter = new SqlDataAdapter(commandString,
connectionString);
```

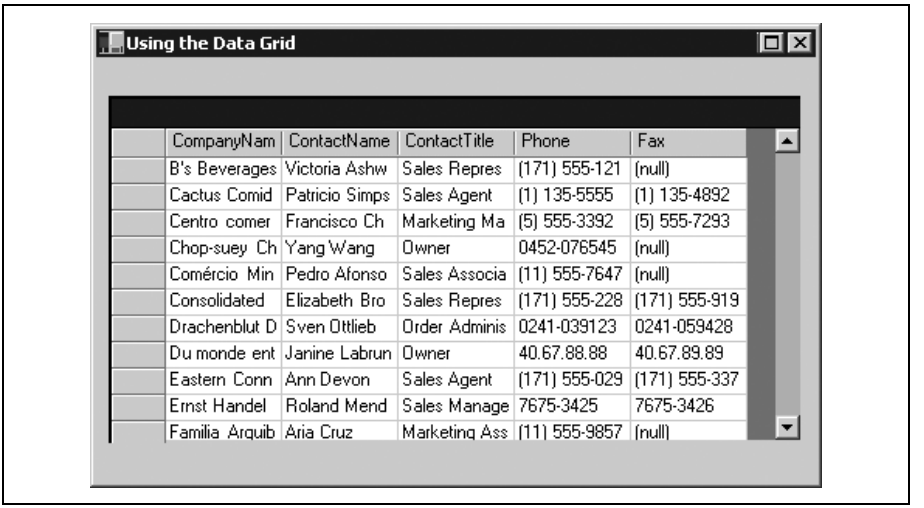

Рис. 14.3. *Элемент управления DataGrid*

На самом деле их значения присваивались объектам `SqlCommand` и `SqlConnection`. Программист может явно создать эти объекты и управлять их свойствами по своему усмотрению.

В следующем примере создаются четыре новых члена класса:

```
private System.Data.SqlClient.SqlConnection myConnection;
private System.Data.DataSet myDataSet;
private System.Data.SqlClient.SqlCommand myCommand;
private System.Data.SqlClient.SqlDataAdapter DataAdapter;
```

Подключение устанавливается созданием экземпляра `SqlConnection` с соответствующей строкой:

```
string connectionString = "server=(local)\\NetSDK;" +
    "serverTrusted_Connection=localhost; uid=sa; pwd=yes;
database=northwind"; myConnection = new
System.Data.Sql.SqlConnection(connectionString);
```

После этого подключение явно открывается:

```
myConnection.Open();
```

После открытия подключения им можно пользоваться неоднократно (что продемонстрировано в следующем примере). Кроме того, при необходимости можно использовать поддержку транзакций.

Теперь явно создадим объект `DataSet` и установим одно из его свойств:

```
myDataSet = new System.Data.DataSet();
myDataSet.CaseSensitive=true;
```

Значение `true` свойства `CaseSensitive` указывает на то, что сравнение строк в объектах `DataTable` производится с учетом регистра символов.

Далее создадим объект SqlCommand и передадим ему объект Connection и текст команды:

```
myCommand = new System.Data.SqlClient.SqlCommand()
myCommand.Connection=myConnection;
myCommand.CommandText = "Select * from Customers";
```

В заключение создадим объект SqlDataAdapter, которому присвоим только что созданный и настроенный объект SqlCommand. С помощью таблицы, в которой выполняется поиск, сообщим объекту DataSet, как он должен отображать столбцы, а объекту SqlDataAdapter велим заполнить объект DataSet:

```
DataAdapter = new System.Data.SqlClient.SqlDataAdapter();
DataAdapter.SelectCommand= myCommand;
DataAdapter.TableMappings.Add("Table","Customers");
DataAdapter.Fill(myDataSet);
```

Все готово для заполнения элемента DataGrid:

```
dataGrid1.DataSource = myDataSet.Tables["Customers"].DefaultView;
```

(На этот раз использовано имя элемента управления, предлагаемое по умолчанию.)

Полный исходный текст рассматриваемой программы содержится в примере 14.4.

Пример 14.4. Настройка объекта DataSet

```
namespace ProgrammingCSharpWindows.Form
{
   using System;
   using System.Drawing;
   using System.Collections;
   using System.ComponentModel;
   using System.Windows.Forms;
   using System.Data;
   using System.Data.SqlClient;

   public class ADOForm1 : System.Windows.Forms.Form
   {
      private System.ComponentModel.Container components;
      private System.Windows.Forms.DataGrid dataGrid1;

      // private System.Data.ADO.ADOConnection myConnection;
      private System.Data.SqlClient.SqlConnection myConnection;
      private System.Data.DataSet myDataSet;
      private System.Data.SqlClient.SqlCommand myCommand;
      private System.Data.SqlClient.SqlDataAdapter DataAdapter;

      public ADOForm1(  )
      {
         InitializeComponent(  );
```

```
   // создать объект соединения и открыть его
   string connectionString = "server=(local)\\NetSDK;" +
      "Trusted_Connection=yes; database=northwind";
   myConnection = new
      System.Data.SqlClient.SqlConnection(connectionString);
   myConnection.Open(  );

   // создать DataSet и установить свойства
   myDataSet = new System.Data.DataSet(  );
   myDataSet.CaseSensitive=true;

   // создать SqlCommand и передать ему соединение и команду select
   myCommand = new System.Data.SqlClient.SqlCommand(  );
   myCommand.Connection=myConnection;
   myCommand.CommandText = "Select * from Customers";

   // создать DataAdapter, передать ему объект SQL-команды
   // и установить способ отображения таблицы
   DataAdapter = new System.Data.SqlClient.SqlDataAdapter(  );
   DataAdapter.SelectCommand= myCommand;
   DataAdapter.TableMappings.Add("Table","Customers");

   // приказать объекту DataAdapter заполнить DataSet
   DataAdapter.Fill(myDataSet);

   // отобразить его в DataGrid
   dataGrid1.DataSource=
      myDataSet.Tables["Customers"].DefaultView;
}

protected override void Dispose(bool disposing)
{
   if (disposing)
   {
       if (components == null)
       {
           components.Dispose(  );
       }
   }
   base.Dispose(disposing);
}

private void InitializeComponent(  )
{
   this.components =
            new System.ComponentModel.Container (  );
   this.dataGrid1 = new System.Windows.Forms.DataGrid (  );
   dataGrid1.BeginInit (  );
   dataGrid1.Location = new System.Drawing.Point (24, 32);
   dataGrid1.Size = new System.Drawing.Size (480, 408);
   dataGrid1.DataMember = "";
   dataGrid1.TabIndex = 0;
   this.Text = "Using the Data Grid";
   this.AutoScaleBaseSize =
```

```
                    new System.Drawing.Size (5, 13);
            this.ClientSize = new System.Drawing.Size (536, 501);
            this.Controls.Add (this.dataGrid1);
            dataGrid1.EndInit (  );
        }

        public static void Main(string[] args)
        {
            Application.Run(new ADOForm1(  ));
        }
    }
}
```

Результат работы этой программы представлен на рис. 14.4. Теперь у программиста гораздо больше возможностей влиять на элемент DataGrid.

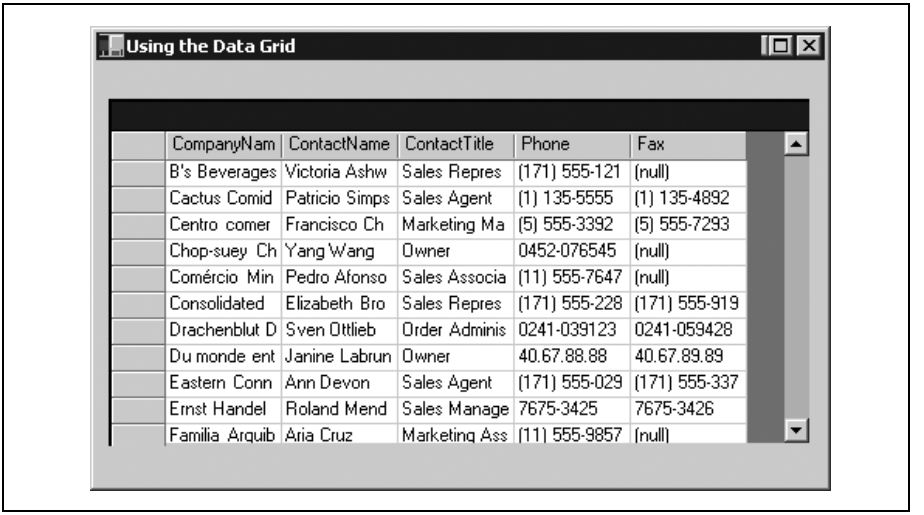

Рис. 14.4. *Прямой контроль над элементом DataGrid*

Объединение таблиц данных

Теперь, когда проделана вся работа по настройке объекта DataSet, будет гораздо проще построить элемент управления DataGrid, отражающий отношения между двумя или несколькими таблицами. Предположим, например, что требуется проанализировать все заказы, сделанные заказчиками в течение некоторого периода.

В основе архитектуры реляционных баз данных лежит идея, что одна таблица имеет определенные отношения с другими таблицами. Отношение между таблицами Orders и Customers состоит в том, что каждый заказ содержит идентификатор заказчика (CustomerID), являющийся *внешним ключом* в таблице Orders и *первичным ключом* в таб-

лице Customers. Тем самым создано отношение «один-ко-многим», при котором один заказчик может иметь много заказов, но каждый заказ имеет только одного заказчика. Попробуем отразить это отношение в элементе DataGrid.

Модель ADO.NET делает эту задачу довольно простой. За основу возьмем предыдущий пример. На этот раз требуется представить две таблицы, Customers и Orders, а не только Customers, как было в предыдущих случаях. Понадобится один объект DataSet, один объект подключения, два объекта SqlCommand и два объекта SqlDataAdapter.

Объект SqlDataAdapter для таблицы Customers создается точно так же, как и в предыдущем примере. После этого создадим вторую команду и адаптер для таблицы Orders:

```
myCommand2 = new System.Data.SqlClient.SqlCommand();
DataAdapter2 = new System.Data.SqlClient.SqlDataAdapter();
myCommand2.Connection = myConnection;
myCommand2.CommandText = "SELECT * FROM Orders";
```

Обратите внимание, что объект DataAdapter2 использует то же подключение, что и объект DataAdapter. Естественно, что текст второй команды отличается от текста первой, так как поиск должен выполняться в другой таблице.

Теперь свяжем второй объект SqlDataAdapter с только что созданной второй командой и отобразим Orders на его таблицу. Теперь можно заполнять объект DataSet информацией из второй таблицы:

```
DataAdapter2.SelectCommand = myCommand2;
DataAdapter2.TableMappings.Add ("Table", "Orders");
DataAdapter2.Fill(myDataSet);
```

В итоге получился один объект DataSet с двумя таблицами внутри. В принципе, можно вывести на экран любую из них или сразу обе, но в этом примере планируется нечто большее. Между двумя таблицами существует отношение, и именно оно будет представлено на экране. К сожалению, объект DataSet не будет осведомлен об этом отношении, пока объект DataRelation не будет явно создан и добавлен к объекту DataSet.

Начнем с объявления объекта DataRelation:

```
System.Data.DataRelation dataRelation;
```

Этот объект представляет собой отношение в базе данных, установленное между полями Customers.CustomerID и Orders.CustomerID. Для моделирования этого отношения понадобится пара объектов типа DataColumn:

```
System.Data.DataColumn dataColumn1;
System.Data.DataColumn dataColumn2;
```

Каждому из этих объектов необходимо присвоить столбец таблицы из объекта DataSet:

```
dataColumn1 = myDataSet.Tables["Customers"].Columns["CustomerID"];
dataColumn2 = myDataSet.Tables["Orders"].Columns["CustomerID"];
```

Теперь все готово для создания объекта DataRelation, конструктору которого следует передать имя отношения и два объекта DataColumn:

```
dataRelation =
 new System.Data.DataRelation("CustomersToOrders", dataColumn1, dataColumn2);
```

Добавим созданный объект отношения к объекту DataSet:

```
myDataSet.Relations.Add(dataRelation);
```

Далее создадим объект DataViewManager, обеспечивающий представление объекта DataSet в объекте DataGrid, и присвоим это представление свойству DataGrid.DataSource:

```
DataViewManager DataSetView = myDataSet.DefaultViewManager;
dataGrid1.DataSource = DataSetView;
```

И наконец, поскольку объект DataGrid содержит несколько таблиц, необходимо сообщить ему, какая таблица является «родителем», иначе говоря, той таблицей, которая относится к остальным по принципу «один-ко-многим». С этой целью устанавливается свойство DataMember элемента DataGrid:

```
dataGrid1.DataMember= "Customers";
```

Полный исходный текст обсуждаемой программы приведен в примере 14.5.

Пример 14.5. Элемент управления DataGrid с двумя таблицами

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace ProgrammingCSharpWindows.Form
{
   using System.Data.SqlClient;

   public class ADOForm1 : System.Windows.Forms.Form
   {
      private System.ComponentModel.Container components;
      private System.Windows.Forms.DataGrid dataGrid1;

      // private System.Data.ADO.ADOConnection myConnection;
      private System.Data.SqlClient.SqlConnection myConnection;
      private System.Data.DataSet myDataSet;
```

```
private System.Data.SqlClient.SqlCommand myCommand;
private System.Data.SqlClient.SqlCommand myCommand2;
private System.Data.SqlClient.SqlDataAdapter DataAdapter;
private System.Data.SqlClient.SqlDataAdapter DataAdapter2;

public ADOForm1()
{
   InitializeComponent();

   // создать подключение
   string connectionString =
      "server=Neptune; uid=sa; pwd=oWenmEany; database=northwind";
   myConnection = new
      System.Data.SqlClient.SqlConnection(connectionString);
   myConnection.Open();

   // создать набор данных
   myDataSet = new System.Data.DataSet();
   myDataSet.CaseSensitive=true;

   // выдать команду и заполнить объект DataSet
   // информацией из первой таблицы
   myCommand = new System.Data.SqlClient.SqlCommand();
   myCommand.Connection=myConnection;
   myCommand.CommandText = "Select * from Customers";
   DataAdapter = new System.Data.SqlClient.SqlDataAdapter();
   DataAdapter.SelectCommand= myCommand;
   DataAdapter.TableMappings.Add("Table","Customers");
   DataAdapter.Fill(myDataSet);

   // выдать команду и заполнить объект DataSet
   // информацией из второй таблицы
   myCommand2 = new System.Data.SqlClient.SqlCommand();
   DataAdapter2 = new System.Data.SqlClient.SqlDataAdapter();
   myCommand2.Connection = myConnection;
   myCommand2.CommandText = "SELECT * FROM Orders";
   DataAdapter2.SelectCommand = myCommand2;
   DataAdapter2.TableMappings.Add ("Table", "Orders");
   DataAdapter2.Fill(myDataSet);

   // установить отношение между таблицами
   System.Data.DataRelation dataRelation;
   System.Data.DataColumn dataColumn1;
   System.Data.DataColumn dataColumn2;
   dataColumn1 = myDataSet.Tables["Customers"].Columns["CustomerID"];
   dataColumn2 = myDataSet.Tables["Orders"].Columns["CustomerID"];

   dataRelation = new System.Data.DataRelation(
      "CustomersToOrders",
      dataColumn1,
      dataColumn2);

   // добавить объект отношения в набор данных
   myDataSet.Relations.Add(dataRelation);
```

```
            // установить представление, обозначить родительскую
            // таблицу и вывести информацию на экран
            DataViewManager DataSetView = myDataSet.DefaultViewManager;
            dataGrid1.DataSource = DataSetView;
            dataGrid1.DataMember = "Customers";
         }

         public override void Dispose()
         {
            base.Dispose();
            components.Dispose();
         }

         private void InitializeComponent()
         {
            this.components = new System.ComponentModel.Container ();
            this.dataGrid1 = new System.Windows.Forms.DataGrid ();
            dataGrid1.BeginInit ();
            //@this.TrayHeight = 0;
            //@this.TrayLargeIcon = false;
            //@this.TrayAutoArrange = true;
            dataGrid1.Location = new System.Drawing.Point (24, 32);
            dataGrid1.Size = new System.Drawing.Size (480, 408);
            dataGrid1.DataMember = "";
            dataGrid1.TabIndex = 0;
            this.Text = "ADOFrm1";
            this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
            this.ClientSize = new System.Drawing.Size (536, 501);
            this.Controls.Add (this.dataGrid1);
            dataGrid1.EndInit ();
         }

         public static void Main(string[] args)
         {
            Application.Run(new ADOForm1());
         }
      }
   }
```

Результат просто поразительный. На рис. 14.5 показан элемент управления DataGrid, в котором выделен один заказчик. Открыта ссылка CustomersToOrders для заказчика с идентификатором CACTU.

Если по ней щелкнуть, откроются все заказы этого заказчика, что изображено на рис. 14.6.

Изменение записей в базе данных

До сих пор выполнялось лишь чтение информации из базы данных, но никакие действия с записями не производились. С помощью модели ADO.NET программист может добавлять записи в таблицу, изменять в ней существующие записи, а также удалять их.

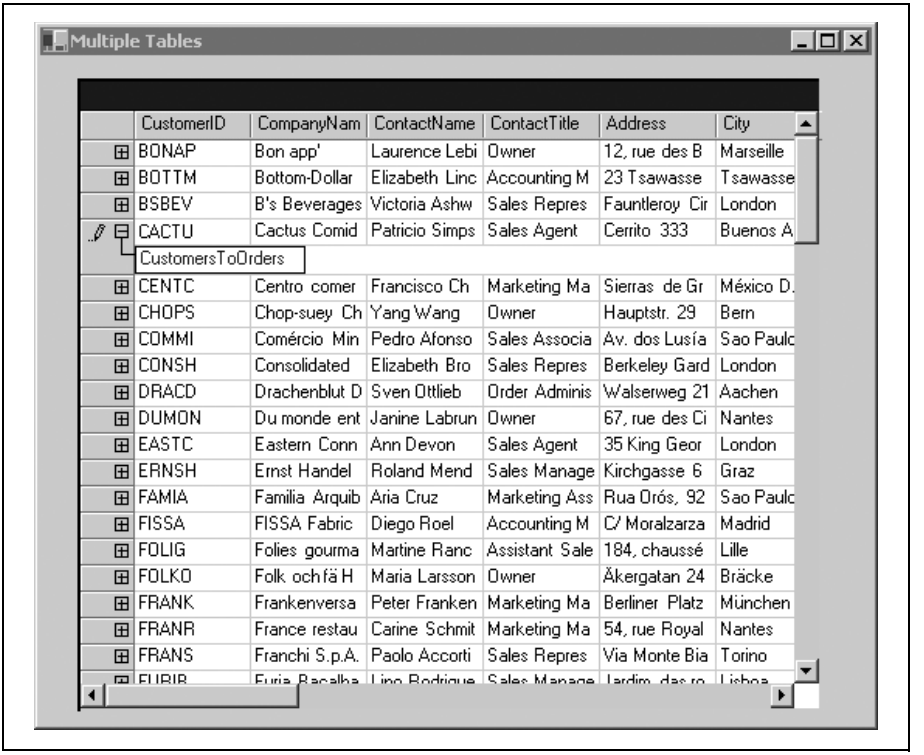

Рис. 14.5. *Таблица заказчиков. Открыта ссылка CustomersToOrders для заказчика с идентификатором CACTU*

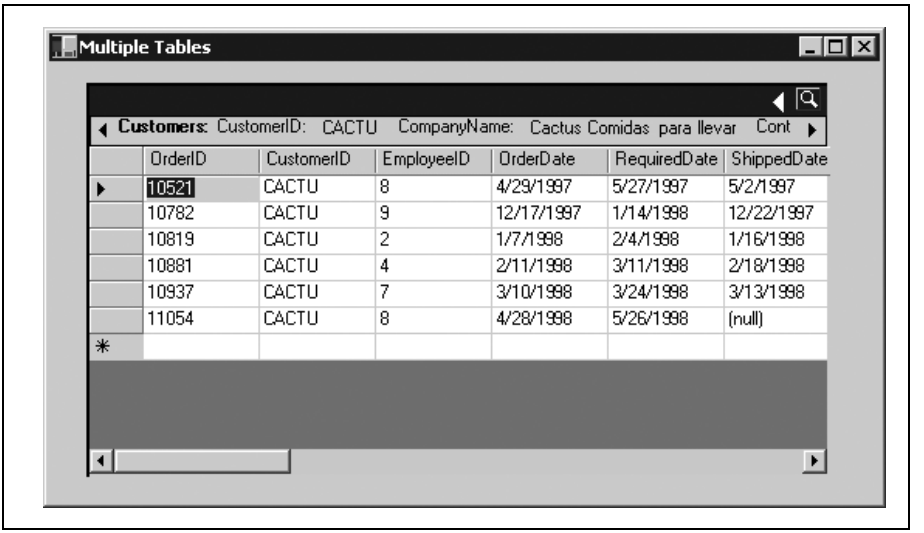

Рис. 14.6. *Все заказы одного заказчика*

В типичном случае работа с базой данных включает в себя следующие шаги:

1. Заполнение таблиц объекта DataSet с помощью хранимой процедуры или запроса SQL.
2. Вывод данных из различных объектов DataTable, хранящихся в объекте DataSet. Это делается либо путем привязки данных к элементам управления, либо циклическим перебором строк в таблицах.
3. Изменение данных в отдельных объектах DataTable за счет добавления, изменения или удаления объектов DataRow.
4. Вызов метода GetChanges() для создания второго объекта DataSet, в котором хранятся только изменения.
5. Проверка наличия ошибок во втором объекте DataSet. С этой целью проверяется его свойство HasErrors. Если ошибки обнаружены, следует проверить одноименные свойства всех объектов DataTable в объекте DataSet. Если в какой-либо таблице присутствуют ошибки, следует вызвать метод GetErrors() соответствующего объекта DataTable и получить от него массив объектов DataRow. У каждой строки (то есть объекта DataRow) нужно проанализировать свойство RowError. Оно содержит специфическую информацию, позволяющую устранить ошибку.
6. Слияние двух объектов DataSet.
7. Вызов метода Update() объекта DataAdapter и передача ему второго (измененного) объекта DataSet.
8. Вызов метода AcceptChanges() объекта DataSet для подтверждения изменений либо вызов метода RejectChanges() для их отмены.

Эта процедура предоставляет программисту полный контроль над процедурой внесения изменений в данные, причем возможные ошибки распознаются уже на ранней стадии.

В следующем примере будет создано диалоговое окно, выводящее содержимое таблицы Customers из базы данных Northwind. Цель этой программы состоит в демонстрации технологии обновления, добавления и удаления табличных записей. Как всегда, автор стремился к максимальному упрощению программы и не включил в пример проверку ошибок и обработку исключений, что, конечно, было бы недопустимо в реальной коммерческой программе.

На рис. 14.7 изображена не очень красивая, но вполне работоспособная форма, созданная автором для проведения экспериментов с моделью ADO.NET.

Эта форма состоит из окна списка (lbCustomers), кнопки для обновления информации (btnUpdate), связанного с ней текстового поля (txtCustomerName) и кнопки удаления записей (btnDelete). В форме присутствуют еще шесть текстовых полей, имеющих отношение к кнопке создания новой записи (btnNew). Эти текстовые поля соответствуют восьми

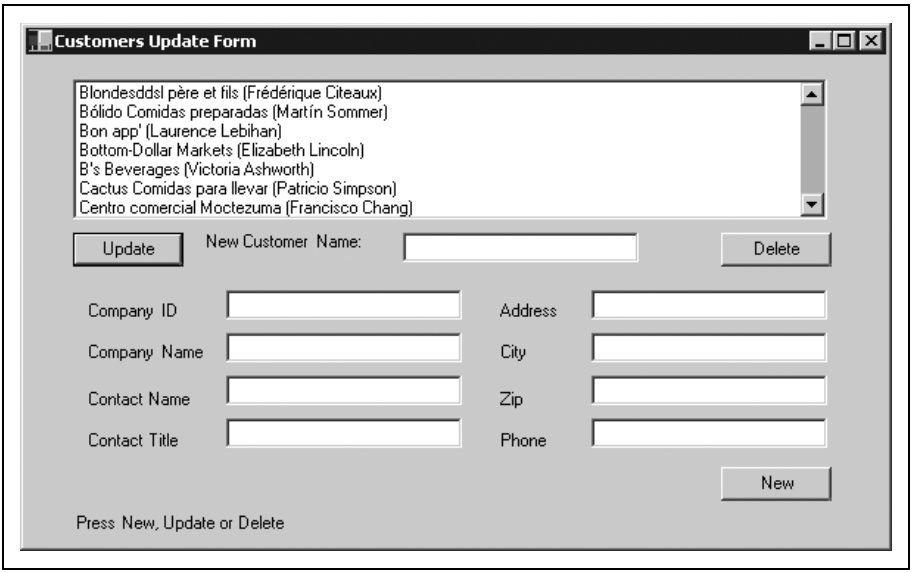

Рис. 14.7. *Форма для экспериментов с ADO*

столбцам таблицы Customers базы данных Northwind. В нижней части формы присутствует статический текст (`lblMessage`), позволяющий выводить сообщения пользователю (в данный момент сообщение гласит: «Press New, Update, or Delete»).

Доступ к данным

Сначала создадим закрытые переменные для объектов `DataAdapter`, `DataSet` и `DataTable`:

```
private SqlDataAdapter dataAdapter;
private DataSet dataSet;
private DataTable dataTable;
```

Они позволят ссылаться на соответствующие объекты из различных методов класса. Затем создадим строки связи и команды, которые позволят извлечь из базы данных нужную таблицу:

```
string connectionString = "server=(local)\\NetSDK;" +
    "serverTrusted_Connection=localhost; uid=sa; pwd=yes;
     database=northwind";
string commandString = "Select * from Customers";
```

Эти строки передаются в качестве аргументов конструктору `SqlDataAdapter`:

```
DataAdapter dataAdapter = new SqlDataAdapter(commandString, connectionString);
```

С `DataAdapter` может быть ассоциировано четыре SQL-комманды. Прямо сейчас у нас есть только одна: `dataAdapter.SelectCommand`. Метод `Ini-`

tializeCommands() создает три оставшиеся: InsertCommand, UpdateCommand и DeleteCommand и использует метод AddParms для связывания столбцов данных из SQL-команд со столбцами в модифицируемых строках:

```
private void AddParms(SqlCommand cmd, params string[] cols) {
  // Добавить каждый параметр
foreach (String column in cols) {
    cmd.Parameters.Add("@" + column, SqlDbType.Char, 0, column);
  }
}
```

InitializeCommands() создает каждую SQL-команду по очереди, используя заполнители, соответствующие аргументам столбца, передаваемым в AddParm():

```
private void InitializeCommands( )
{

   // Повторно используем соединение от SelectCommand.
   SqlConnection connection =(SqlConnection)
dataAdapter.SelectCommand.Connection;

// Создаем явную, повторно используемую команду insert
   dataAdapter.InsertCommand = connection.CreateCommand(  );
   dataAdapter.InsertCommand.CommandText =
   "Insert into customers " +
      "(CustomerId, CompanyName, ContactName, ContactTitle, " +
      " Address, City, PostalCode, Phone) " +
      "values(@CustomerId, @CompanyName, @ContactName, " +
      "   @ContactTitle, @Address, @City, @PostalCode, @Phone)";
   AddParms(dataAdapter.InsertCommand,
     "CustomerId", "CompanyName", "ContactName", "ContactTitle",
     "Address", "City", "PostalCode", "Phone");

   // Создаем явную команду update
   dataAdapter.UpdateCommand = connection.CreateCommand(  );
   dataAdapter.UpdateCommand.CommandText = "update Customers " +
     "set CompanyName = @CompanyName where CustomerID = @CustomerId";
   AddParms(dataAdapter.UpdateCommand, "CompanyName", "CustomerID");

   // Создаем явную команду delete
   dataAdapter.DeleteCommand = connection.CreateCommand(  );
   dataAdapter.DeleteCommand.CommandText =
     "delete from customers where customerID = @CustomerId";
   AddParms(dataAdapter.DeleteCommand, "CustomerID");
}
```

DataAdapter использует эти три команды для внесения изменений в таблицу при вызове метода Update().

Но вернемся к конструктору. Теперь создадим объект DataSet и заполним его с помощью только что созданного адаптера данных:

```
dataSet = new DataSet();
dataAdapter.Fill(DataSet,"Customers");
```

Чтобы вывести содержимое таблицы на экран, вызовем метод PopulateLB(), закрытый метод, заполняющий список содержимым таблицы из DataSet:

```
dataTable = dataSet.Tables[0];
lbCustomers.Items.Clear();
foreach (DataRow dataRow in dataTable.Rows)
{
    lbCustomers.Items.Add(
        dataRow["CompanyName"] + " (" + dataRow["ContactName"] + ")" );
}
```

Обновление записи

Итак, форма находится на экране, и в ней нужно изменить какую-то запись. Выделите запись и введите новое имя заказчика в самое верхнее текстовое поле. После щелчка по кнопке Update имя будет занесено в выбранную запись.

Первое, что надо сделать в программе, – получить строку, которую пользователь хочет изменить:

```
DataRow targetRow = dataTable.Rows[lbCustomers.SelectedIndex];
```

Объявим объект типа DataRow и инициализируем его ссылкой на строку из коллекции Rows объекта DataTable, соответствующую выделенному элементу списка. Вспомните, что объект DataTable был объявлен в предыдущем разделе как переменная класса и заполнен с помощью метода PopulateLB().

Выведем название фирмы, запись о которой будет изменена:

```
lblMessage.Text = "Updating " +  targetRow["CompanyName"];
Application.DoEvents();
```

Вызов статического метода DoEvents() класса Application заставляет приложение обрабатывать сообщения системы Windows и осуществлять вывод на экран. Если бы этой строчки не было, вычислительный поток занял бы все время процессора, и сообщения не выводились бы на экран вплоть до окончания обработки щелчка кнопки.

Чтобы перевести строку в режим редактирования, вызовем метод BeginEdit() объекта DataRow. Тем самым будут приостановлены вызовы каких-либо событий, связанных со строкой, что позволит отредактировать сразу несколько строк без проверки корректности (впрочем, в этом примере подобная проверка отсутствует). Вообще, хороший стиль программирования предписывает начинать внесение изменений в объект DataRows вызовом метода BeginEdit(), а заканчивать – вызовом EndEdit():

```
targetRow.BeginEdit();
targetRow["CompanyName"] = txtCustomerName.Text;
targetRow.EndEdit();
```

Здесь выполняется редактирование поля CompanyName в объекте targetRow, которому присваивается текст из элемента управления txtCustomerName. В результате поле CompanyName в соответствующей строке получит значение, введенное пользователем в текстовое поле.

Обратите внимание, что нужное поле индексируется по его имени. В рассматриваемом примере это имя соответствует имени столбца в базе данных, что, вообще говоря, не обязательно. При создании объекта DataSet можно было изменить имена столбцов с помощью метода TableMappings().

Отредактировав поле строки, можно приступать к проверке на отсутствие ошибок. Прежде всего извлечем из объекта DataSet все внесенные изменения (в данном случае только одно). Для этой цели вызывается метод GetChanges(), которому следует передать перечислимое значение DataRowState, указывающее на то, что требуется получить только строки, в которые были внесены изменения. Метод GetChanges() возвратит новый объект DataSet:

```
DataSet dataSetChanged = dataSet.GetChanges(DataRowState.Modified);
```

Теперь выполним собственно проверку ошибок. Для упрощения программы введем некий флаг, показывающий, что ошибок нет. Если ошибки обнаружены, не станем их исправлять, а просто установим для флага значение false и не будем вносить изменения:

```
bool okayFlag = true;
if (dataSetChanged.HasErrors)
{
   okayFlag = false;
   string msg = "Error in row with customer ID ";

   foreach (DataTable theTable in dataSetChanged.Tables)
   {
      if (theTable.HasErrors)
      {
         DataRow[] errorRows = theTable.GetErrors();

         foreach (DataRow theRow in errorRows)
         {
            msg = msg + theRow["CustomerID"];
         }
      }
   }
   lblMessage.Text = msg;
}
```

Первая проверка относится к набору данных в целом. Если его свойство HasErrors равно true, значит, ошибки присутствуют. Присвоим

флагу okayFlag значение false и продолжим локализацию ошибки. Просмотрим в цикле все таблицы нового объекта DataSet (в рассматриваемом примере у него только одна таблица). Если в какой-нибудь таблице есть ошибка, будет возвращен массив ее строк, содержащих ошибки (массив errorRows).

Далее нужно просмотреть в цикле элементы этого массива, поочередно исправляя ошибки. В данном примере достаточно вывести сообщение в диалоговом окне, но в реальной ситуации исправление ошибок потребовало бы взаимодействия с пользователем.

Если после проверки свойства HasErrors флаг okayFlag имеет значение true, значит, ошибок нет и можно вносить изменения в базу данных:

```
if (okayFlag)
{
    dataAdapter.Update(dataSetChanged, "Customers");
```

В этом коде объект DataAdapter создаст команду, необходимую для обновления самой базы данных. Теперь обновим сообщение на экране:

```
lblMessage.Text = "Updated"+"targetRow["Company Name"];
Application.DoEvents();
```

Теперь необходимо указать объекту DataSet, чтобы он принял изменения, а затем заполнить список обновленной информацией из объекта DataSet:

```
dataSet.AcceptChanges();
PopulateLB();
```

Если бы флаг okayFlag имел значение false, это свидетельствовало бы о наличии ошибок. В таком случае достаточно просто отменить изменения:

```
else
     dataSet.RejectChanges();
```

Удаление записей

Обработка щелчка по кнопке Delete еще проще. Вначале получим выделенную строку:

```
DataRow targetRow  = dataTable.Rows[lbCustomers.SelectedIndex];
```

и сформируем сообщение об удалении строки:

```
string msg = targetRow["CompanyName"] + " deleted. ";
```

Его не следует выводить на экран до окончания операции, но подготовить надо сейчас, иначе будет слишком поздно!

Теперь можно пометить строку на удаление:

```
targetRow.Delete();
```

Вызов метода AcceptChanges() для объекта DataSet влечет за собой вызовы одноименных методов для всех таблиц этого объекта. А это, в свою очередь, приведет к вызовам методов AcceptChanges() для каждой строки таблицы. Иными словами, одно обращение к методу dataSet.AcceptChanges() каскадно распространяет его на все таблицы и строки.

Пришло время вызвать Update() и AcceptChanges(), а затем обновить список на экране. В то же время, если эти действия вызовут сбой, строка по-прежнему будет помечена на удаление. При этом выполнение любой другой команды, например вставки (insert), обновления (update) или другого удаления, приведет к тому, что DataAdapter попытается повторить ошибочное удаление, снова не сможет это сделать и прервет выполнение всей накопившейся последовательности команд. Чтобы избежать этой ситуации, следует поместить эти операции в блок try и вызвать RejectChanges() в случае неудачи:

```
// обновление базы данных
try
{
  dataAdapter.Update(dataSet,"Customers");
  dataSet.AcceptChanges(  );
  // Обновление списка
  PopulateLB(  );

  // информируем пользователя
  lblMessage.Text = msg;
  Application.DoEvents(  );
}
catch (SqlException ex)
{
  dataSet.RejectChanges(  );
  MessageBox.Show(ex.Message);
}
```

При удалении записей из таблицы Customers может быть вызвано исключение, если база данных наложила на запись ограничения, связанные с целостностью данных. В частности, если у заказчика есть заказы в таблице Orders, запись о нем нельзя удалять, пока не удалены записи о заказах. Чтобы справиться с подобной проблемой, в следующем примере будут созданы новые записи таблицы Customers, которые можно будет удалить беспрепятственно.

Создание новых записей

Для создания новой записи пользователь должен заполнить поля формы и нажать кнопку New. Возникнет событие btnNew_Click, которое связывается с обработчиком btnNew_Click:

```
btnNew.Click += new System.EventHandler (this.btnNew_Click);
```

В обработчике события будет вызываться метод DataTable.NewRow(), запрашивающий у таблицы новый объект DataRow:

```
DataRow newRow = dataTable.NewRow();
```

Это очень элегантное решение, поскольку новая строка, возвращенная объектом DataTable, содержит все необходимые объекты DataColumn. Достаточно будет заполнить поля строки значениями, взятыми из текстовых полей формы:

```
newRow["CustomerID"] = txtCompanyID.Text;
newRow["CompanyName"] = txtCompanyName.Text;

newRow["ContactName"] = txtContactName.Text;
newRow["ContactTitle"] = txtContactTitle.Text;
newRow["Address"] = txtAddress.Text;
newRow["City"] = txtCity.Text;
newRow["PostalCode"] = txtZip.Text;
newRow["Phone"] = txtPhone.Text;
```

Теперь, когда строка заполнена, следует добавить ее в таблицу:

```
dataTable.Rows.Add(newRow);
```

Эта таблица содержится в объекте DataSet, поэтому следует сообщить объекту DataAdapter, что он должен обновить базу данных и принять изменения:

```
dataAdapter.Update(dataSet,"Customers");
dataSet.AcceptChanges();
```

Теперь обновим пользовательский интерфейс:

```
lblMessage.Text = "Updated!";
Application.DoEvents();
```

Заново заполним список формы так, чтобы он содержал добавленную строку, а текстовые поля очистим:

```
PopulateLB();
ClearFields();
```

ClearFields() — это закрытый метод, присваивающий всем текстовым полям пустые строки. Этот метод и программа целиком приведены в примере 14.6.

Пример 14.6. Обновление, удаление и добавление записей

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;
namespace ProgrammingCSharpWindows.Form
{
```

```
public class ADOForm1 : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components;
   private System.Windows.Forms.Label label9;
   private System.Windows.Forms.TextBox txtPhone;
   private System.Windows.Forms.Label label8;
   private System.Windows.Forms.TextBox txtContactTitle;
   private System.Windows.Forms.Label label7;
   private System.Windows.Forms.TextBox txtZip;
   private System.Windows.Forms.Label label6;
   private System.Windows.Forms.TextBox txtCity;
   private System.Windows.Forms.Label label5;
   private System.Windows.Forms.TextBox txtAddress;
   private System.Windows.Forms.Label label4;
   private System.Windows.Forms.TextBox txtContactName;
   private System.Windows.Forms.Label label3;
   private System.Windows.Forms.TextBox txtCompanyName;
   private System.Windows.Forms.Label label2;
   private System.Windows.Forms.TextBox txtCompanyID;
   private System.Windows.Forms.Label label1;
   private System.Windows.Forms.Button btnNew;
   private System.Windows.Forms.TextBox txtCustomerName;
   private System.Windows.Forms.Button btnUpdate;
   private System.Windows.Forms.Label lblMessage;
   private System.Windows.Forms.Button btnDelete;
   private System.Windows.Forms.ListBox lbCustomers;

   // DataSet, DataAdapter и DataTable - члены класса,
   // соответственно, они видны из других методов-членов класса.
   private SqlDataAdapter dataAdapter;
   private DataSet dataSet;
   private DataTable dataTable;

   public ADOForm1(  )
   {
      InitializeComponent(  );

      string connectionString = "server=(local)\\NetSDK;" +
        "Trusted_Connection=yes; database=northwind";
      string commandString = "Select * from Customers";
      dataAdapter =
         new SqlDataAdapter(commandString, connectionString);

      InitializeCommands(  );

      dataSet = new DataSet(  );
      dataAdapter.Fill(dataSet,"Customers");
      PopulateLB(  );
   }

   private void AddParms(SqlCommand cmd, params string[] cols) {
     // Добавляем каждый параметр
     foreach (String column in cols) {
       cmd.Parameters.Add("@" + column, SqlDbType.Char, 0, column);
     }
```

```
}

private void InitializeCommands(  )
{

   // Повторно используем соединение от SelectCommand
   SqlConnection connection =
       (SqlConnection) dataAdapter.SelectCommand.Connection;

   // Создаем явную, повторно используемую команду insert
   dataAdapter.InsertCommand = connection.CreateCommand(  );
   dataAdapter.InsertCommand.CommandText =
   "Insert into customers " +
      "(CustomerId, CompanyName, ContactName, ContactTitle, " +
      " Address, City, PostalCode, Phone) " +
      "values(@CustomerId, @CompanyName, @ContactName, " +
      "   @ContactTitle, @Address, @City, @PostalCode, @Phone)";
   AddParms(dataAdapter.InsertCommand,
     "CustomerId", "CompanyName", "ContactName", "ContactTitle",
     "Address", "City", "PostalCode", "Phone");

   // Создаем явную команду update
   dataAdapter.UpdateCommand = connection.CreateCommand(  );
   dataAdapter.UpdateCommand.CommandText = "update Customers " +
     "set CompanyName = @CompanyName where CustomerID = @CustomerId";
   AddParms(dataAdapter.UpdateCommand, "CompanyName", "CustomerID");

   // Создаем явную команду delete
   dataAdapter.DeleteCommand = connection.CreateCommand(  );
   dataAdapter.DeleteCommand.CommandText =
     "delete from customers where customerID = @CustomerId";
   AddParms(dataAdapter.DeleteCommand, "CustomerID");
}

// Заполняем список столбцами из таблицы Customers
private void PopulateLB(  )
{
   dataTable = dataSet.Tables[0];
   lbCustomers.Items.Clear(  );
   foreach (DataRow dataRow in dataTable.Rows)
   {
      lbCustomers.Items.Add(
         dataRow["CompanyName"] + " (" +
         dataRow["ContactName"] + ")" );
   }

}

protected override void Dispose(bool disposing)
{
   if (disposing)
   {
       if (components == null)
       {
           components.Dispose(  );
       }
```

```
        }
        base.Dispose(disposing);
    }

    private void InitializeComponent(  )
    {
        this.components = new System.ComponentModel.Container (  );
        this.txtCustomerName = new System.Windows.Forms.TextBox (  );
        this.txtCity = new System.Windows.Forms.TextBox (  );
        this.txtCompanyID = new System.Windows.Forms.TextBox (  );
        this.lblMessage = new System.Windows.Forms.Label (  );
        this.btnUpdate = new System.Windows.Forms.Button (  );
        this.txtContactName = new System.Windows.Forms.TextBox (  );
        this.txtZip = new System.Windows.Forms.TextBox (  );
        this.btnDelete = new System.Windows.Forms.Button (  );
        this.txtContactTitle = new System.Windows.Forms.TextBox (  );
        this.txtAddress = new System.Windows.Forms.TextBox (  );
        this.txtCompanyName = new System.Windows.Forms.TextBox (  );
        this.label5 = new System.Windows.Forms.Label (  );
        this.label6 = new System.Windows.Forms.Label (  );
        this.label7 = new System.Windows.Forms.Label (  );
        this.label8 = new System.Windows.Forms.Label (  );
        this.label9 = new System.Windows.Forms.Label (  );
        this.label4 = new System.Windows.Forms.Label (  );
        this.lbCustomers = new System.Windows.Forms.ListBox (  );
        this.txtPhone = new System.Windows.Forms.TextBox (  );
        this.btnNew = new System.Windows.Forms.Button (  );
        this.label1 = new System.Windows.Forms.Label (  );
        this.label2 = new System.Windows.Forms.Label (  );
        this.label3 = new System.Windows.Forms.Label (  );
        //@this.TrayHeight = 0;
        //@this.TrayLargeIcon = false;
        //@this.TrayAutoArrange = true;
        txtCustomerName.Location = new System.Drawing.Point (256, 120);
        txtCustomerName.TabIndex = 4;
        txtCustomerName.Size = new System.Drawing.Size (160, 20);
        txtCity.Location = new System.Drawing.Point (384, 245);
        txtCity.TabIndex = 15;
        txtCity.Size = new System.Drawing.Size (160, 20);
        txtCompanyID.Location = new System.Drawing.Point (136, 216);
        txtCompanyID.TabIndex = 7;
        txtCompanyID.Size = new System.Drawing.Size (160, 20);
        lblMessage.Location = new System.Drawing.Point (32, 368);
        lblMessage.Text = "Press New, Update or Delete";
        lblMessage.Size = new System.Drawing.Size (416, 48);
        lblMessage.TabIndex = 1;
        btnUpdate.Location = new System.Drawing.Point (32, 120);
        btnUpdate.Size = new System.Drawing.Size (75, 23);
        btnUpdate.TabIndex = 0;
        btnUpdate.Text = "Update";
        btnUpdate.Click += new System.EventHandler (this.btnUpdate_Click);
        txtContactName.Location = new System.Drawing.Point (136, 274);
```

```
txtContactName.TabIndex = 11;
txtContactName.Size = new System.Drawing.Size (160, 20);
txtZip.Location = new System.Drawing.Point (384, 274);
txtZip.TabIndex = 17;
txtZip.Size = new System.Drawing.Size (160, 20);
btnDelete.Location = new System.Drawing.Point (472, 120);
btnDelete.Size = new System.Drawing.Size (75, 23);
btnDelete.TabIndex = 2;
btnDelete.Text = "Delete";
btnDelete.Click += new System.EventHandler (this.btnDelete_Click);
txtContactTitle.Location = new System.Drawing.Point (136, 303);
txtContactTitle.TabIndex = 12;
txtContactTitle.Size = new System.Drawing.Size (160, 20);
txtAddress.Location = new System.Drawing.Point (384, 216);
txtAddress.TabIndex = 13;
txtAddress.Size = new System.Drawing.Size (160, 20);
txtCompanyName.Location = new System.Drawing.Point (136, 245);
txtCompanyName.TabIndex = 9;
txtCompanyName.Size = new System.Drawing.Size (160, 20);
label5.Location = new System.Drawing.Point (320, 252);
label5.Text = "City";
label5.Size = new System.Drawing.Size (48, 16);
label5.TabIndex = 14;
label6.Location = new System.Drawing.Point (320, 284);
label6.Text = "Zip";
label6.Size = new System.Drawing.Size (40, 16);
label6.TabIndex = 16;
label7.Location = new System.Drawing.Point (40, 312);
label7.Text = "Contact Title";
label7.Size = new System.Drawing.Size (88, 16);
label7.TabIndex = 28;
label8.Location = new System.Drawing.Point (320, 312);
label8.Text = "Phone";
label8.Size = new System.Drawing.Size (56, 16);
label8.TabIndex = 20;
label9.Location = new System.Drawing.Point (120, 120);
label9.Text = "New Customer Name:";
label9.Size = new System.Drawing.Size (120, 24);
label9.TabIndex = 22;
label4.Location = new System.Drawing.Point (320, 224);
label4.Text = "Address";
label4.Size = new System.Drawing.Size (56, 16);
label4.TabIndex = 26;
lbCustomers.Location = new System.Drawing.Point (32, 16);
lbCustomers.Size = new System.Drawing.Size (512, 95);
lbCustomers.TabIndex = 3;
txtPhone.Location = new System.Drawing.Point (384, 303);
txtPhone.TabIndex = 18;
txtPhone.Size = new System.Drawing.Size (160, 20);
btnNew.Location = new System.Drawing.Point (472, 336);
btnNew.Size = new System.Drawing.Size (75, 23);
```

```
    btnNew.TabIndex = 25;
    btnNew.Text = "New";
    btnNew.Click += new System.EventHandler (this.btnNew_Click);
    label1.Location = new System.Drawing.Point (40, 224);
    label1.Text = "Company ID";
    label1.Size = new System.Drawing.Size (88, 16);
    label1.TabIndex = 6;
    label2.Location = new System.Drawing.Point (40, 252);
    label2.Text = "Company Name";
    label2.Size = new System.Drawing.Size (88, 16);
    label2.TabIndex = 8;
    label3.Location = new System.Drawing.Point (40, 284);
    label3.Text = "Contact Name";
    label3.Size = new System.Drawing.Size (88, 16);
    label3.TabIndex = 10;
    this.Text = "Customers Update Form";
    this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
    this.ClientSize = new System.Drawing.Size (584, 421);
    this.Controls.Add (this.label9);
    this.Controls.Add (this.txtPhone);
    this.Controls.Add (this.label8);
    this.Controls.Add (this.txtContactTitle);
    this.Controls.Add (this.label7);
    this.Controls.Add (this.txtZip);
    this.Controls.Add (this.label6);
    this.Controls.Add (this.txtCity);
    this.Controls.Add (this.label5);
    this.Controls.Add (this.txtAddress);
    this.Controls.Add (this.label4);
    this.Controls.Add (this.txtContactName);
    this.Controls.Add (this.label3);
    this.Controls.Add (this.txtCompanyName);
    this.Controls.Add (this.label2);
    this.Controls.Add (this.txtCompanyID);
    this.Controls.Add (this.label1);
    this.Controls.Add (this.btnNew);
    this.Controls.Add (this.txtCustomerName);
    this.Controls.Add (this.btnUpdate);
    this.Controls.Add (this.lblMessage);
    this.Controls.Add (this.btnDelete);
    this.Controls.Add (this.lbCustomers);
}

// обработка щелчка по кнопке new
protected void btnNew_Click (object sender, System.EventArgs e)
{

    // create a new row, populate it
    DataRow newRow = dataTable.NewRow(  );
    newRow["CustomerID"]   = txtCompanyID.Text;
    newRow["CompanyName"]  = txtCompanyName.Text;
    newRow["ContactName"]  = txtContactName.Text;
```

```
    newRow["ContactTitle"] = txtContactTitle.Text;
    newRow["Address"]      = txtAddress.Text;
    newRow["City"]         = txtCity.Text;
    newRow["PostalCode"]   = txtZip.Text;
    newRow["Phone"]        = txtPhone.Text;

    // добавить в таблицу новую строку
    dataTable.Rows.Add(newRow);

    // обновить базу данных
    try
    {
      dataAdapter.Update(dataSet,"Customers");
      dataSet.AcceptChanges(  );

      // сообщить пользователю
      lblMessage.Text = "Updated!";
      Application.DoEvents(  );

      // заполнить список заново
      PopulateLB(  );
      // clear all the text fields
      ClearFields(  );
    }
    catch (SqlException ex)
    {
      dataSet.RejectChanges(  );
      MessageBox.Show(ex.Message);
    }

  }

  // очистить все текстовые поля
  private void ClearFields(  )
  {
    txtCompanyID.Text = "";
    txtCompanyName.Text = "";
    txtContactName.Text = "";
    txtContactTitle.Text = "";
    txtAddress.Text = "";
    txtCity.Text = "";
    txtZip.Text = "";
    txtPhone.Text = "";
  }

  // обработка щелчка по кнопке update
  protected void btnUpdate_Click (object sender, System.EventArgs e)
  {
    // получить выбранную строку
    DataRow targetRow  = dataTable.Rows[lbCustomers.SelectedIndex];
    // сообщить пользователю
    lblMessage.Text = "Updating " +  targetRow["CompanyName"];
    Application.DoEvents(  );

    // редактировать строку
```

```
    targetRow.BeginEdit(  );
    targetRow["CompanyName"] = txtCustomerName.Text;
    targetRow.EndEdit(  );

    // получить каждую измененную строку
    DataSet dataSetChanged = dataSet.GetChanges(DataRowState.Modified);

    // проверить, что во всех измененных строках нет ошибок
    bool okayFlag = true;
    if (dataSetChanged.HasErrors)
    {
       okayFlag = false;
       string msg = "Error in row with customer ID ";

       // проверить каждую таблицу в измененном DataSet
       foreach (DataTable theTable in dataSetChanged.Tables)
       {
          // если в таблице есть ошибки - найти ошибочные строки
          if (theTable.HasErrors)
          {
             // получить ошибочные строки
             DataRow[] errorRows = theTable.GetErrors(  );

             // перебрать ошибки и исправить
             // (в нашем случае - только показать)
             foreach (DataRow theRow in errorRows)
             {
                msg = msg + theRow["CustomerID"];
             }
          }
       }
       lblMessage.Text = msg;
    }
    // if we have no errors
    if (okayFlag)
    {

       // обновить базу данных
       dataAdapter.Update(dataSetChanged,"Customers");

       // сообщить пользователю
       lblMessage.Text = "Updated " +  targetRow["CompanyName"];
       Application.DoEvents(  );

       // принять изменения и заполнить заново список
       dataSet.AcceptChanges(  );
       PopulateLB(  );
    }

    else  // если имели место ошибки, отказаться от изменений
       dataSet.RejectChanges(  );
}

// обработать щелчок по кнопке delete
protected void btnDelete_Click (object sender, System.EventArgs e)
{
```

```
            // получить выделенную строку
            DataRow targetRow  = dataTable.Rows[lbCustomers.SelectedIndex];

            // подготовить сообщение пользователю
            string msg = targetRow["CompanyName"] + " deleted. ";

            // удалить выделенную строку
            targetRow.Delete(  );

            // обновить базу данных
            try
            {
              dataAdapter.Update(dataSet,"Customers");
              dataSet.AcceptChanges(  );

              // заново заполнить список
              PopulateLB(  );

              // сообщить пользователю
              lblMessage.Text = msg;
              Application.DoEvents(  );
            }
            catch (SqlException ex)
            {
              dataSet.RejectChanges(  );
              MessageBox.Show(ex.Message);
            }

         }

         public static void Main(string[] args)
         {
            Application.Run(new ADOForm1(  ));
         }
      }
   }
```

На рис. 14.8 изображена заполненная форма непосредственно перед щелчком по кнопке New.

На рис. 14.9 показано, как выглядит форма сразу после добавления новой записи.

Обратите внимание на то, что новая запись добавлена в конец списка, а текстовые поля пусты.

Модель ADO.NET и технология XML

В этой главе были продемонстрированы способы обработки данных, знакомые пользователям ADO. Кроме того, было показано, как ADO.NET поддерживает своими библиотеками классов работу с базами данных. Однако картина была бы неполной без упоминания о том, что модель ADO.NET оказывает полную поддержку технологии XML. Самое интересное, что ADO.NET позволяет представлять содержимое

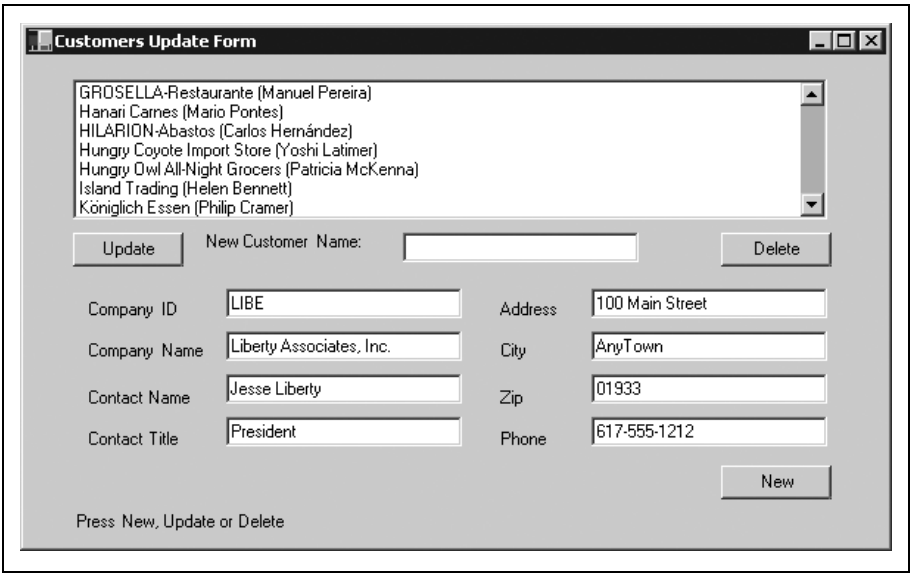

Рис. 14.8. *Подготовка к добавлению новой записи*

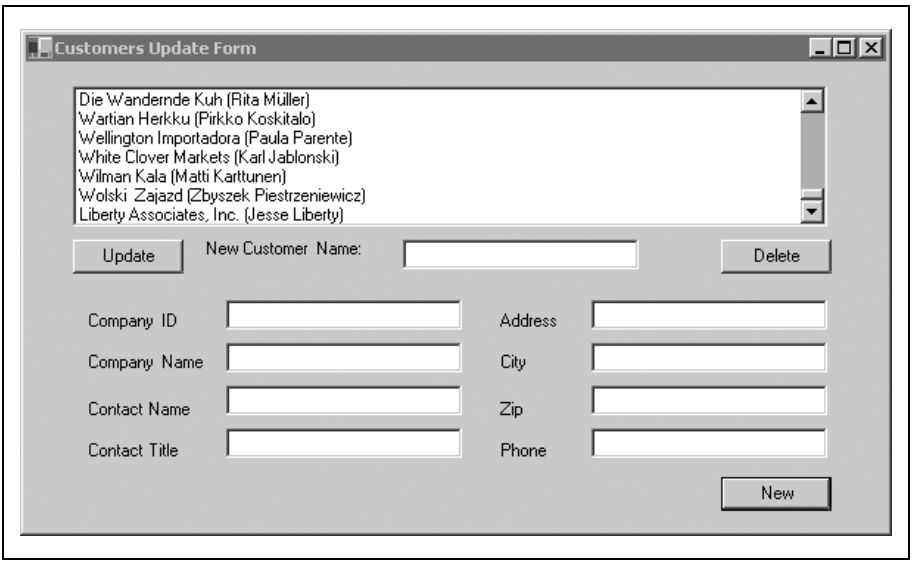

Рис. 14.9. *Форма после добавления новой записи*

набора данных либо в виде коллекции таблиц, как в этой главе, либо в виде XML-документа.

Тесная интеграция ADO.NET и XML, а также создание соответствующих приложений не являются предметом обсуждения данной книги. Полная информация по этой теме содержится в справочнике «.NET Framework SDK Reference».

15

Создание веб-приложений с помощью Web Forms

Все большее число программистов отходит в настоящее время от создания традиционных Windows- и клиент-серверных приложений и пишет веб-приложения даже в тех случаях, когда приложение заведомо не использует клиент-серверную архитектуру. Есть много очевидных аргументов в пользу такого решения. Например, программисту не приходится много времени тратить на разработку пользовательского интерфейса, так как Internet Explorer или Netscape Navigator сделают за него почти всю работу. Другим, возможно, более важным достоинством веб-приложений является быстрота, простота и дешевизна распространения новых версий. Когда автор, еще до появления Всемирной паутины, работал в фирме, обслуживавшей одну компьютерную сеть, стоимость распространения новых версий оценивалась по формуле: 1 миллион долларов на количество дискет (читатель еще помнит дискеты?). Себестоимость распространения веб-приложений практически равна нулю. Третьим достоинством веб-приложений является распределенная обработка. Имея такое приложение, гораздо проще реализовать обработку данных на стороне сервера. В Сети существует множество стандартных протоколов (например, HTTP, HTML и XML), облегчающих построение n-уровневых приложений.

Технология, принятая в платформе .NET для построения веб-приложений (и динамических веб-сайтов), носит имя ASP.NET и описывается в книге «Programming ASP.NET», O'Reilly, 2002.[1] В своих пространствах имен `System.Web` и `System.Web.UI` она содержит богатейший набор типов, необходимых при создании веб-приложений. В этой главе

[1] Дж. Либерти, Д. Гурвиц «Программирование на ASP.NET». – Пер. с англ. – СПб: Символ-плюс, II квартал 2003 года.

особое внимание уделяется области пересечения ASP.NET и C#, а именно Web Forms.

Среда Web Forms поддерживает ту же технику быстрой разработки приложений (RAD), что и Windows Forms. Программист может перетаскивать элементы управления на форму и писать код поддержки, либо встроенный, либо на страницах поддержки. Однако есть и отличие. В среде Web Forms приложение распространяется через веб-сервер, а пользователь взаимодействует с программой через стандартный броузер.

Среда Web Forms

Среда Web Forms реализует модель программирования, в которой веб-страницы динамически генерируются на веб-сервере и передаются броузеру через Интернет. В определенном смысле эта среда является развитием технологии ASP, позволяя объединить ASP с традиционным программированием.

В среде Web Forms разработчик создает HTML-страницы со статическим содержимым и пишет программу C# для вывода динамического содержимого. Программа C# выполняется на сервере, и созданные ею данные интегрируются со статической HTML-страницей, в результате чего получается веб-страница. Броузеру отсылается стандартный HTML-код.

Веб-формы (приложения, созданные с помощью Web Forms) могут работать с любым броузером, если он может работать с версией HTML, используемой сервером при создании страниц. Логика работы веб-формы может быть воспроизведена на любом языке платформы .NET. В этой книге, конечно же, используется C#, хотя разработчики ASP, привыкшие работать на языке VBScript, возможно, остановят свой выбор на VB.NET.

Как и в случае форм Windows, веб-формы *можно* создавать в редакторе Блокнот (или любом другом), а не в среде Visual Studio .NET. Многие программисты так и делают, но все-таки Visual Studio .NET *значительно* облегчает процесс разработки и тестирования веб-форм.

В среде Web Forms пользовательский интерфейс разделен на две части: визуальная составляющая и логика, которая находится «за кулисами». Здесь уместна аналогия со средой Windows Forms, описанной в главе 13, но в среде Web Forms страница с пользовательским интерфейсом и программа находятся в разных файлах.

Страница с интерфейсом хранится в файле, имеющем расширение *.aspx*. Логика (программная поддержка) этой страницы находится в отдельном файле поддержки, представляющем собой исходный файл на языке C#. При обращении к форме вызывается класс поддержки, динамически создающий код HTML, посылаемый броузеру клиента. Этот класс активно пользуется типами Web Forms, расположенными в

пространстве имен `System.Web` и `System.Web.UI`, принадлежащими библиотеке классов платформы .NET Framework (.NET Framework Class Library, сокращенно FCL).

Нет ничего проще программирования веб-форм в среде Visual Studio .NET. Открываем форму, перетаскиваем на нее необходимые элементы управления и пишем методы обработки событий. Готово! Веб-приложение написано.

С другой стороны, даже в среде Visual Studio .NET создание надежного полноценного веб-приложения может оказаться непростой задачей. Среда Web Forms предлагает богатый выбор элементов пользовательского интерфейса. Количество и сложность элементов управления значительно возросли за последние годы, соответственно возросли и ожидания пользователей относительно внешнего вида и поведения веб-приложений.

Кроме всего прочего, веб-приложения по своей природе являются распределенными. В типичном случае клиент и сервер расположены в разных местах. В большинстве веб-приложений при создании интерфейса приходится учитывать такие реалии, как время ожидания в Сети, ширина полосы пропускания и производительность сетевого сервера. Передача информации от клиента к хост-компьютеру и обратно может длиться несколько секунд.

События, связанные с веб-формами

Веб-формы управляются событиями. *Событие* (*event*) – это объект, инкапсулирующий идею «произошло нечто важное». Событие возникает (или *вызывается*), когда пользователь нажимает кнопку, выделяет элемент в списке или как-то иначе взаимодействует с интерфейсом. События могут быть вызваны и системой в начале и конце какого-либо процесса. Например, программа открывает файл на чтение, а система вызывает событие по окончании загрузки файла в память.

Метод, реагирующий на событие, называется *обработчиком события*. Обработчики событий пишутся на языке C# на странице поддержки и связываются с элементами управления на HTML-странице с помощью атрибутов этих элементов.

Обработчики событий являются делегатами (см. главу 12). Существует соглашение, по которому обработчик события в ASP.NET возвращает значение типа `void` и имеет два параметра. Первый параметр представляет собой объект, вызвавший событие. Второй, называемый *аргументом события*, содержит информацию, специфичную для события, если таковая имеется. В большинстве случаев типом аргумента события является `EventArgs`, у которого нет никаких свойств. У некоторых элементов управления аргумент события может иметь тип, производный от `EventArgs` и обладающий свойствами, специфичными для конкретного события.

Как правило, события, возникающие в веб-приложениях, обрабатываются на сервере, а для этого приходится передавать информацию от клиента серверу и обратно. В технологии ASP.NET предусмотрен лишь ограниченный набор событий, таких как нажатие кнопок и внесение изменений в текстовые поля. Существует ряд событий, которые пользователь вправе ожидать от веб-приложения (в отличие от Windows-приложения) и которые могут возникать неоднократно в процессе взаимодействия с пользователем. В качестве примера можно привести событие «указатель мыши находится на объекте».

Отправляющие и неотправляющие события

Отправляющими (*postback*) называются события, которые приводят к немедленной отправке формой сообщения на сервер. К этим событиям относятся щелчки кнопками мыши, например нажатие на кнопку. С другой стороны, многие события (обычно связанные с изменением состояния элементов управления) считаются *неотправляющими* (*nonpostback*), поскольку они не требуют немедленной отправки сообщения формой. Эти события кэшируются элементом управления до того момента, когда произойдет отправляющее событие. Элементы с неотправляющими событиями можно принудить к отправляющей манере поведения, если присвоить их свойству `AutoPostBack` значение `true`.

Состояние

Состояние веб-приложения – это совокупность текущих значений всех элементов управления и переменных для данного пользователя в данном сеансе. Сама Сеть по природе своей является средой без сохранения состояния. Это означает, что каждая отправка данных на сервер приводит к потере предыдущего состояния, если разработчик не предпримет специальных действий по сохранению этой информации о сеансе. Впрочем, ASP.NET позволяет сохранять состояние пользовательского сеанса.

После отправки страницей сообщения серверу он заново воссоздает страницу и возвращает ее броузеру. В технологии ASP.NET предусмотрен механизм автоматического сохранения состояния элементов управления. Если пользователь получает список и выделяет в нем какой-либо элемент, то это выделение будет сохранено после передачи страницы на сервер и впоследствии восстановлено на клиентском компьютере.

Жизненный цикл веб-формы

Каждый запрос страницы, поступающий на веб-сервер, вызывает цепочку событий, происходящих на сервере. Эти события, от первого до последнего, образуют *жизненный цикл* страницы и всех ее компонентов. Он начинается с запроса страницы, что приводит к ее загрузке. Когда запрос завершен, страница выгружается. На протяжении всего

жизненного цикла задача сервера состоит в передаче соответствующего кода HTML отправившему запрос броузеру. Жизненный цикл страницы отмечен следующими событиями, обработка которых может быть выполнена программистом или оставлена на усмотрение сервера ASP.NET:

Инициализация

Инициализация – это первый этап жизненного цикла любой страницы или элемента управления. Здесь выполняются все настройки, необходимые для выполнения поступившего запроса.

Загрузка ViewState

Свойство `ViewState` элемента управления заполняется необходимой информацией, которая берется из скрытой переменной, используемой для сохранения состояния во время пересылки данных на сервер и обратно. Входная строка, поступившая от скрытой переменной, анализируется страницей, и свойству `ViewState` присваивается соответствующее значение. Оно может быть изменено методом `LoadViewState()`. Это позволяет среде ASP.NET контролировать состояние элемента управления между отдельными загрузками страницы и не допускать его сброс в значение по умолчанию при каждой загрузке.

Обработка полученных данных

На этой стадии обрабатываются данные, посланные на сервер клиентом. Если в результате их обработки потребуется изменить свойство `ViewState`, это будет сделано методом `LoadPostData()`.

Загрузка

Вызывается метод `CreateChildControls()` для создания и инициализации серверных элементов управления, содержащихся в иерархическом списке. Состояние восстанавливается, и элементы управления выводят данные клиента. Программист может изменить поведение формы на этом этапе, обработав событие `Load` в методе `OnLoad()`.

Отправка измененного состояния

Если текущее состояние отличается от предыдущего, метод `RaisePostDataChangedEvent()` вызывает событие, отражающее этот факт.

Обработка отправляющих событий

Обрабатывается событие на стороне клиента, инициирующее отправку данных формы на сервер.

Подготовка к выводу

Данный этап непосредственно предшествует передаче результата броузеру. Фактически это последняя возможность внести изменения в возвращаемую информацию с помощью метода `OnPreRender()`.

Сохранение состояния

Ближе к началу жизненного цикла сохраненное состояние было загружено из скрытой переменной. Теперь оно сохраняется в этой

переменной в виде строкового объекта, который будет отправлен клиенту. Для внесения изменений в эту операцию следует перегрузить метод `SaveViewState()`.

Формирование вывода

На этом этапе собирается возвращаемая броузеру клиента информация. Для внесения изменений в эту операцию следует перегрузить метод `Render()`. При необходимости вызывается метод `CreateChildControls()` для создания и инициализации элементов управления в иерархическом списке серверных элементов управления.

Освобождение ресурсов

Это заключительный этап жизненного цикла. Здесь можно выполнить необходимую «приборку» и освободить дорогостоящие ресурсы, например подключение к базе данных. Прохождение этого этапа можно изменить при помощи метода `Dispose()`.

Создание веб-формы

Чтобы создать простую веб-форму для следующего примера, запустите Visual Studio .NET и откройте новый проект ProgrammingCSharpWeb. Выберите папку Visual C# Projects, поскольку проект будет написан на языке C#, а в качестве типа проекта – ASP.NET Web Application. В поле Name введите имя проекта. В качестве местоположения проекта среда Visual Studio .NET по умолчанию предложит *http://localhost* (рис. 15.1).

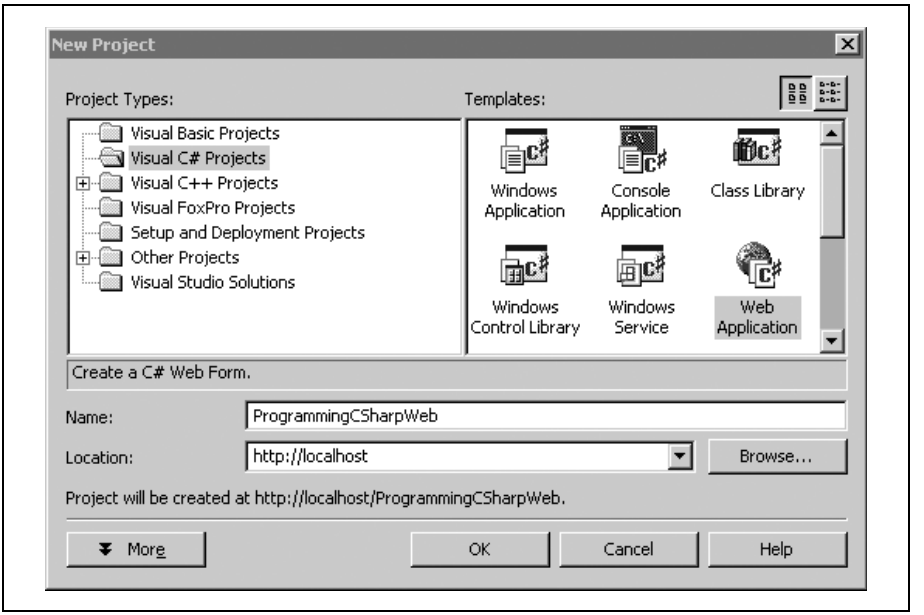

Рис. 15.1. *Создание проекта в окне New Project среды Visual Studio .NET*

Почти все файлы, которые Visual Studio .NET создаст для проекта, будут размещены в указанной по умолчанию папке веб-сайта на компьютере разработчика, например *c:\Inetpub\wwwroot\ProgrammingCSharpWeb*.

В среде Visual Studio .NET *решением* (*solution*) называется набор проектов, каждый из которых представляет собой либо библиотеку динамической компоновки (DLL), либо исполняемый файл (EXE). Все проекты создаются в контексте какого-нибудь решения, а сами решения управляются файлами с расширением *.sln* или *.suo*.

Файлы решений и другие файлы, имеющие отношение к Visual Studio .NET, хранятся в каталоге *<диск>\Documents* and *Settings\<имя пользователя>\My Documents\Visual Studio Projects* (где *<диск>* и *<имя пользователя>* зависят от конкретного компьютера).

Чтобы воспользоваться инструментами Web Forms, необходимо заранее установить на компьютере сервер IIS и серверные расширения FrontPage. Для конфигурации серверных расширений FrontPage откройте Internet Service Manager и щелкните правой кнопкой мыши по имени веб-сайта. Выберите команду меню All Tasks → Configure Server Extensions. Более подробную информацию можно найти на сайте *http://www.microsoft.com*.

Когда приложение будет создано, Visual Studio .NET разместит в проекте несколько файлов. Сама веб-форма будет находиться в файле *WebForm1.aspx*, который содержит только код HTML. Второй столь же важный файл, *WebForm1.aspx.cs*, хранит в себе программу на языке C#, связанную с данной формой. Это файл поддержки.

Обратите внимание, что файл поддержки *отсутствует* в окне Solution Explorer. Чтобы увидеть его, необходимо в окне Visual Studio .NET щелкнуть по форме правой кнопкой мыши и выбрать команду View Code в появившемся контекстном меню. Теперь можно переключаться с файла формы *WebForm1.aspx* на файл *WebForm1.aspx.cs* и обратно. Просматривая файл формы, можно выбрать либо режим Design, либо режим HTML (соответствующие кнопки расположены в самом низу окна редактора формы). Режим Design позволяет перетаскивать элементы управления на форму, а режим HTML – просматривать и редактировать сам код HTML.

Рассмотрим ASPX- и CS-файлы, созданные средой Visual Studio .NET, более внимательно. Начнем с переименования *WebForm1.aspx* в *HelloWeb.aspx*. Для этого следует закрыть файл и щелкнуть правой кнопкой мыши на его имени в окне Solution Explorer. Выберите команду Rename и введите *HelloWeb.aspx*. Переименовав файл, откройте его и просмот-

рите код. Кстати, CS-файл тоже будет переименован, его новое имя – *HelloWeb.aspx.cs*.

При создании веб-приложения среда Visual Studio .NET создает множество заготовок, позволяющих разработчику приступить к программированию веб-формы (пример 15.1).

Пример 15.1. Заготовка кода веб-формы

```
<%@ Page language="c#"
  Codebehind="HelloWeb.aspx.cs"
  AutoEventWireup="false"
  Inherits="ProgrammingCSharpWeb.WebForm1" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <head>
    <title>WebForm1</title>
    <meta name="GENERATOR"
      Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
  </head>
  <body MS_POSITIONING="GridLayout">

    <form id="Form1" method="post" runat="server">

    </form>

  </body>
</html>
```

Здесь представлена типичная заготовка кода HTML. Оригинальными являются только первые строки с кодом ASP.NET:

```
<%@ Page language="c#"
Codebehind="HelloWeb.aspx.cs"
AutoEventWireup="false"
Inherits="ProgrammingCSharpWeb.WebForm1" %>
```

Атрибут `language` сообщает, что страница поддержки написана на языке C#. Атрибут `Codebehind` сообщает, что эта страница хранится в файле *HelloWeb.cs*, а атрибут `Inherits` указывает, что страница произведена от класса `WebForm1`. Этот класс объявлен в файле *HelloWeb.cs*.

```
public class WebForm1 : System.Web.UI.Page
```

Как следует из программы C#, класс `WebForm1` является потомком класса `System.Web.UI.Page`, в котором определены свойства, методы и события, общие для всех серверных страниц.

Вернемся к HTML-представлению файла *HelloWeb.aspx*. Видно, что форма определена в теле страницы с помощью стандартного тега языка HTML:

```
<form id="Form1" method="post" runat="server">
```

Среда Web Forms полагает, что для взаимодействия с пользователем программисту потребуется как минимум одна форма, которая и создается при открытии нового проекта. Атрибут `runat="server"` является принципиально важным для серверной обработки. Любой тег, содержащий этот атрибут, считается элементом, который должен быть выполнен на сервере в среде ASP.NET.

После создания пустой веб-формы программист, скорее всего, захочет дописать код веб-страницы. Переключившись в режим HTML, он может добавить прямо в файл любой сценарий и код HTML, как это делается в классическом случае с ASP. Чтобы вывести на экран приветствие и местное время, добавим в тело HTML-страницы следующую строчку:

```
Hello World! It is now <% = DateTime.Now.ToString() %>
```

Ограничители <% и %> имеют здесь тот же смысл, что и в традиционных ASP-страницах. Они содержат в себе код на языке программирования (в данном случае на C#). Знак равенства (=), стоящий непосредственно после открывающего тега (<%), предписывает среде ASP.NET вывести значение на экран, как это делает метод `Response.Write()`. С таким же успехом можно было написать:

```
Hello World! It is now
<% Response.Write(DateTime.Now.ToString()); %>
```

Выполните эту страницу, нажав комбинацию клавиш <Ctrl>+<F5> (или сохраните ее и затем загрузите в броузере). Появится окно, изображенное на рис. 15.2.

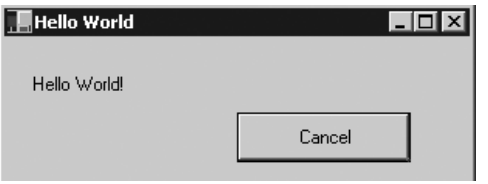

Рис. 15.2. *Результат обработки файла HelloWeb.aspx*

Добавление элементов управления

Серверные элементы управления могут быть добавлены к веб-форме двумя способами: вручную (за счет того, что на HTML-странице пишется необходимый код) или путем перетаскивания их из окна Toolbox

на форму, в режиме Design. Пусть, например, программист решил предоставить пользователю три переключателя для выбора грузоперевозчика (из таблицы Shippers базы данных Northwind). В разделе <form> HTML-страницы нужно написать следующий код:

```
<asp:RadioButton GroupName="Shipper" id="Airborne"
    text = "Airborne Express" Checked="True" runat="server">
</asp:RadioButton>
<asp:RadioButton GroupName="Shipper" id="UPS"
    text = "United Parcel Service" runat="server">
</asp:RadioButton>
<asp:RadioButton GroupName="Shipper" id="Federal"
    text = "Federal Express" runat="server">
</asp:RadioButton>
```

Теги <asp> объявляют серверные элементы управления из среды ASP.NET. Они заменяются стандартными HTML-тегами, когда сервер обрабатывает страницу. После запуска приложения броузер выведет на экран группу из трех переключателей, среди которых можно выбрать только один.

Тот же результат можно получить ценой гораздо меньших усилий, если перетащить переключатели в окне Visual Studio .NET из окна Toolbox на форму (рис. 15.3).

Разработчик может добавлять элементы управления в одном из двух режимов. По умолчанию установлен режим GridLayout. Когда элементы добавляются в этом режиме, броузер выводит их на экран, используя абсолютное позиционирование (с помощью горизонтальной и вертикальной координат). Разработчик отвечает за создание HTML-кода, обеспечивающего их позиционирование.

Альтернативный режим называется FlowLayout. В этом режиме элементы выводятся последовательно сверху вниз, как в документе Microsoft Word. Переключение из одного режима в другой осуществляется в среде Visual Studio .NET с помощью свойства pageLayout документа.

Среда Web Forms предлагает программисту два вида серверных элементов управления. Первый вид – это серверные HTML-элементы, называемые также веб-элементами. Они представляют собой стандартные HTML-элементы, помеченные атрибутом runat="server".

Альтернативой веб-элементам являются серверные элементы среды ASP.NET, для краткости именуемые ASP-элементами. ASP-элементы разработаны для замены стандартных HTML-элементов. Они образуют более стройную объектную модель с непротиворечивыми названиями атрибутов. Например, стандартные HTML-элементы позволяют реализовать вывод информации множеством различных способов:

```
<input type = "radio">
<input type="checkbox">
<input type="button">
```

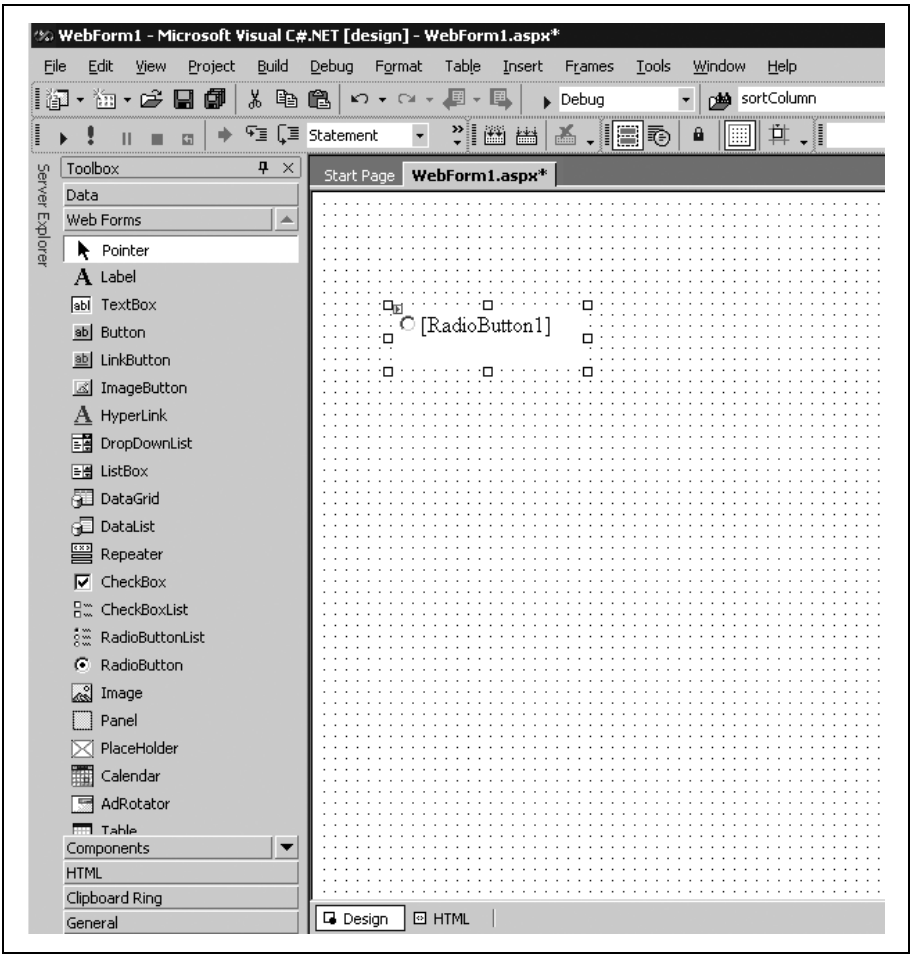

Рис. 15.3. *Перетаскивание элементов на веб-форму*

```
<input type="text">
<textarea>
```

Все эти элементы управления ведут себя по-разному и имеют разные атрибуты. В ASP.NET предпринята попытка нормализовать набор элементов управления и упорядочить использование атрибутов для всей объектной модели. Элементам HTML, представленным в предыдущем фрагменте программы, соответствуют следующие ASP-элементы:

```
<asp:RadioButton>
<asp:CheckBox>
<asp:Button>
<asp:TextBox rows="1">
<asp:TextBox rows="5">
```

Далее в этой главе обсуждаются исключительно ASP-элементы.

Привязка данных

Существует целый ряд технологий, декларирующих возможность связывания элементов управления с данными. Целью такой операции является автоматическая реакция элемента на внесение изменений в данные. Однако, как часто говорил Роки Булльвинклю: «Но этот фокус никогда не получается».[1] Элементы управления с возможностью привязки данных, как правило, предоставляют весьма ограниченные возможности контроля над их внешним видом и поведением, а их производительность ниже всякой критики. Разработчики ASP.NET задались целью разрешить эти проблемы и предоставить программистам комплект надежных элементов управления с возможностью привязки данных, облегчающих внесение изменений в эти данные и вывод их на экран без снижения производительности и качества пользовательского интерфейса.

В предыдущем разделе в форме были жестко установлены три переключателя, по одному на каждого грузоперевозчика из базы данных Northwind. Это не самое лучшее решение. Если перевозчики сменятся, придется перепрограммировать элементы управления. В этом разделе будет показано, как создавать их динамически и затем связывать с базой данных.

Создавать переключатели, получающие информацию из базы данных, предпочтительнее, поскольку на этапе проектирования неизвестно, какой текст должен сопровождать переключатели и даже каково их количество. Для реализации задуманного потребуется элемент управления `RadioButtonList`. Он позволяет создавать переключатели динамически: программист указывает их имена и значения, а ASP.NET заботится обо всем остальном.

Удалите группу переключателей, расположенную на форме, и перетащите на их место элемент `RadioButtonList`. В окне `Properties` переименуйте его в `rbl1`.

Установка начальных значений свойств

Программирование в среде Web Forms является управляемым событиями. В большинстве случаев эти события инициализируются пользователем. Например, когда пользователь щелкает по кнопке, возникает событие `Button_Click`.

Самым важным изначальным событием является `Page_Load`, возникающее всякий раз, когда загружается веб-форма. По окончании ее загрузки нужно будет присвоить переключателям значения, взятые из базы данных. Например, если создается форма для совершения покуп-

[1] Роки и Булльвинкль – персонажи мультсериала о приключениях летающей белки и говорящего лося. – *Примеч. науч. ред.*

ки, можно создать переключатель для каждого перевозчика грузов, то есть UPS, FedEx и т. д. Поэтому вызовы методов, создающих переключатели, должны быть помещены в метод, обрабатывающий событие Page_Load.

Свойства должны быть присвоены переключателям только после первой загрузки страницы. Если пользователь щелкнет по кнопке или както иначе пошлет сообщение формы на сервер, то будет неразумно заново читать значения из базы данных после следующей загрузки страницы.

Среда ASP.NET способна отличить самую первую загрузку от всех последующих, происходящих после отправки сообщения формы от клиента серверу. Каждая веб-страница обладает свойством IsPostBack, которое имеет значение true, если страница загружается в ответ на отправку ее сообщения серверу, и false, если она загружается впервые.

Достаточно проверить значение этого свойства и, если оно равно false, прочитать информацию из базы данных, поскольку страница наверняка загружается в первый раз:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {//... }
}
```

Аргументы метода Page_Load() — это нормальные аргументы обработчика события, обсужденные в главе 12.

Подключение к базе данных

Фрагмент программы, устанавливающий связь с базой данных и заполняющий набор записей, уже знаком читателю; он практически идентичен фрагменту программы, приведенному в главе 14. Нет никакой разницы между созданием набора записей для веб-формы и для формы Windows.

Начнем с объявления переменных класса:

```
private System.Data. SqlClient.SqlConnection myConnection;
private System.Data.DataSet myDataSet;
private System.Data. SqlClient.SqlCommand myCommand;
private System.Data.SqlClient.SqlDataAdapter dataAdapter;
```

Как и в главе 14, здесь используются SQL-версии объекта подключения и адаптера данных. Для связи с базой данных Northwind создадим строковую переменную connectionString и воспользуемся ею для создания и открытия объекта SQLConnection:

```
string connectionString = "server=(local)\\NetSDK;+
    " Trusted_Connection=yes; database=northwind";
myConnection = new System.Data. SqlClient.SqlConnection(connectionString);
myConnection.Open();
```

Создадим набор данных и настроим его на обработку запросов с учетом регистра символов:

```
myDataSet = new System.Data.DataSet();
myDataSet.CaseSensitive=true;
```

Далее нужно создать объект SqlCommand и присвоить ему объект подключения и текст команды Select, необходимые для получения идентификатора ShipperID и названия компании-грузоперевозчика. Идентификатор будет служить значением переключателя, а название компании – текстом:

```
myCommand = new System.Data.SQL.SqlCommand();
myCommand.Connection=myConnection;
myCommand.CommandText = "Select ShipperID, CompanyName from Shippers";
```

Создадим объект dataAdapter, присвоим его свойству SelectCommand объект команды и добавим таблицу Shippers к отображению таблиц:

```
dataAdapter = new System.Data.SqlClient.SqlDataAdapter();
dataAdapter.SelectCommand= myCommand;
dataAdapter.TableMappings.Add("Table","Shippers");
```

Наконец, заполним объект dataAdapter результатами запроса:

```
dataAdapter.Fill(myDataSet);
```

Все эти действия практически совпадают с тем, что было в главе 14. Однако сейчас требуется привязать данные к элементу управления RadioButtonList, созданному ранее.

Первый шаг процедуры привязки состоит в установке свойств объекта RadioButtonList. Вначале установим свойство, определяющее способ компоновки переключателей на странице:

```
rbl1.RepeatLayout = System.Web.UI.WebControls.RepeatLayout.Flow;
```

Flow – одно из двух значений перечислимого типа RepeatLayout. Другое, Table, компонует переключатели табличным способом. Далее сообщим объекту RadioButtonList, какие значения из набора данных будут выведены на экран в виде текста (свойство DataTextField) и какое значение будет возвращаться, когда пользователь установит один из трех флажков (свойство DataValueField):

```
rbl1.DataTextField = "CompanyName";
rbl1.DataValueField = "ShipperID";
```

Наконец, сообщим элементу управления, какое представление данных будет использоваться. В рассматриваемом примере воспользуемся предлагаемым по умолчанию представлением таблицы Shippers, принятым по умолчанию:

```
rbl1.DataSource = myDataSet.Tables["Shippers"].DefaultView;
```

Установив свойства элемента управления RadioButtonList, привяжем его к набору данных:

```
rbl1.DataBind();
```

В заключение обязательно нужно выделить один из переключателей; выделим первый:

```
rbl1.Items[0].Selected = true;
```

В этом операторе значение true присваивается свойству Selected первого элемента коллекции Items, которая является свойством объекта RadioButtonList (иначе говоря, выделяется первый переключатель).

Если запустить программу на исполнение или вызвать эту страницу в броузере, то переключатели будут выведены на экран, как это показано на рис. 15.4.

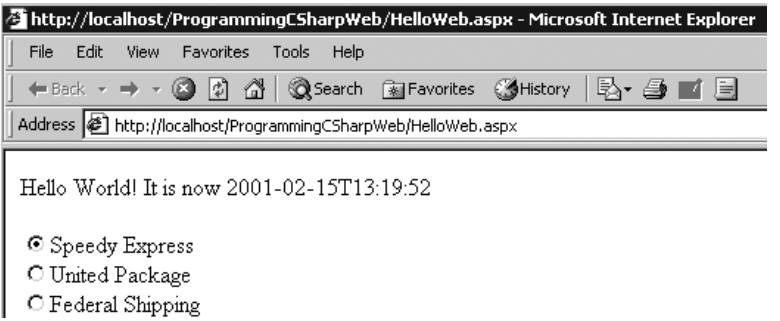

Рис. 15.4. *Элемент управления RadioButtonList*

Изучив исходный код страницы, читатель не найдет в нем элемент RadioButtonList. Вместо него будут проставлены стандартные HTML-теги переключателей с общим атрибутом id. Последнее обстоятельство позволяет броузеру трактовать их как единую группу. Для всех переключателей созданы заголовки, и каждый заголовок помещен в тег :

```
<span id="rbl1" style="...">
<input id="rbl1_0" type="radio" name="rbl1" value="1" checked="checked" />
<label for="rbl1_0">Speedy Express</label>
<br>
<!- оставшиеся кнопки опущены для краткости -->
</span>
```

Этот HTML-код создан сервером, объединившим обработку элемента RadioButtonList в HTML-коде с обработкой страницы поддержки. Когда страница загружается, вызывается метод Page_Load() и заполняется адаптер данных. Присваивание свойству rbl1.DataTextField значения «CompanyName», свойству rbl1.DataValueField значения «ShipperID», а свойству rbl1.DataSource значения, определяющего представление таблицы Shippers по умолчанию, подготавливает список переключате-

лей (то есть элемент RadioButtonList) к созданию переключателей. Вызов метода DataBind() приводит к созданию переключателей на основе информации, полученной от источника данных.

После добавления еще нескольких элементов управления получается законченная форма, пригодная для взаимодействия с пользователем. В частности, следует добавить подходящее к ситуации приветствие «Welcome to Northwind», текстовое поле для имени пользователя, две кнопки (Order и Cancel) для подтверждения и отмены заказа соответственно и текст, поясняющий, что требуется от пользователя: «Please choose the shipper» (Пожалуйста, выберите грузоперевозчика). Окончательный внешний вид формы изображен на рис. 15.5.

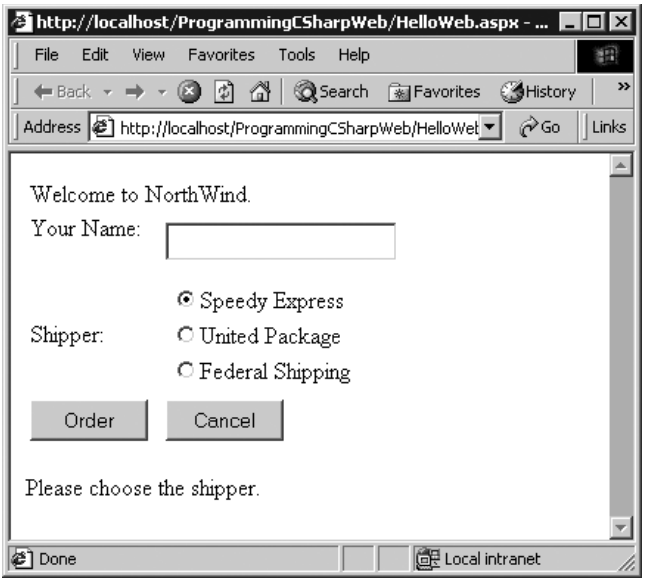

Рис. 15.5. *Законченная форма*

Эта форма не получит премии за дизайн, но она хорошо иллюстрирует некоторые ключевые моменты, связанные с веб-формами.

Автор не встречал разработчика, который не был бы уверен, что может создать безупречный дизайн пользовательского интерфейса. В то же время, автору не приходилось встречать разработчиков, которые действительно были бы в состоянии сделать это. Дизайн графического интерфейса – одна из областей, где все считают себя специалистами (как, например, в преподавании), однако лишь немногим талантливым людям удается добиться в нем серьезных успехов. Про себя автор может сказать: «Я программист, мое дело написать программу, а компоновкой веб-страницы должен заниматься художник».

Полный HTML-код обсуждаемого ASPX-файла приведен в примере 15.2.

Пример 15.2. Файл с веб-формой

```
<%@ Page language="c#"
  Codebehind="HelloWeb.aspx.cs"
  AutoEventWireup="false"
  Inherits="ProgrammingCSharpWeb.WebForm1" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>WebForm1</title>
    <meta name="GENERATOR"
      Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript"
      content="JavaScript">
    <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>

  <body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post" runat="server">

      <asp:Label id="Label1"
        style="Z-INDEX: 101; LEFT: 20px; POSITION: absolute; TOP: 28px"
        runat="server">Welcome to NorthWind.</asp:Label>

      <asp:Label id="Label2"
        style="Z-INDEX: 102; LEFT: 20px; POSITION: absolute; TOP: 67px"
        runat="server">Your Name:</asp:Label>

      <asp:Label id="Label3"
        style="Z-INDEX: 103; LEFT: 20px; POSITION: absolute; TOP: 134px"
        runat="server">Shipper:</asp:Label>

      <asp:Label id="lblFeedBack"
        style="Z-INDEX: 104; LEFT: 20px; POSITION: absolute; TOP: 241px"
        runat="server">Please choose the shipper.</asp:Label>

      <asp:Button id="Order"
        style="Z-INDEX: 105; LEFT: 20px; POSITION: absolute; TOP: 197px"
        runat="server" Text="Order"></asp:Button>

      <asp:Button id="Cancel"
        style="Z-INDEX: 106; LEFT: 128px; POSITION: absolute; TOP: 197px"
        runat="server" Text="Cancel"></asp:Button>

      <asp:TextBox id="txtName"
        style="Z-INDEX: 107; LEFT: 128px; POSITION: absolute; TOP: 64px"
        runat="server"></asp:TextBox>

      <asp:RadioButtonList id="rbl1"
        style="Z-INDEX: 108; LEFT: 112px; POSITION: absolute; TOP: 130px"
        runat="server"></asp:RadioButtonList>
```

```
    </form>
  </body>

</HTML>
```

Элементы <asp:button> будут преобразованы в стандартные HTML-теги <input>. Снова вспомним о достоинствах ASP-элементов: они составляют непротиворечивую объектную модель и к тому же транслируются в стандартный HTML-код, который может быть воспринят любым броузером. Поскольку ASP-элементы помечаются атрибутом runat="server" и имеют атрибут id, к ним можно обращаться из программы, расположенной на сервере, когда потребуется. В примере 15.3 содержится полный текст программы, поддерживающей HTML-форму.

Пример 15.3. Программа поддержки HTML-формы

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ProgrammingCSharpWeb
{
    // конструктор страницы
    public class WebForm1 : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.WebControls.Label Label2;
        protected System.Web.UI.WebControls.Label Label3;
        protected System.Web.UI.WebControls.Label lblFeedBack;
        protected System.Web.UI.WebControls.Button Order;
        protected System.Web.UI.WebControls.Button Cancel;
        protected System.Web.UI.WebControls.TextBox txtName;
        protected System.Web.UI.WebControls.RadioButtonList rbl1;

        private System.Data.SqlClient.SqlConnection myConnection;
        private System.Data.DataSet myDataSet;
        private System.Data.SqlClient.SqlCommand myCommand;
        private System.Data.SqlClient.SqlDataAdapter dataAdapter;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // при первой загрузке страницы получить данные
            // и установить переключатели
            if (!IsPostBack)
            {
                string connectionString = "server=(local)\\NetSDK;" +
```

```
            "Trusted_Connection=yes; database=northwind";
        myConnection = new System.Data.SqlClient.SqlConnection(
            connectionString);
        myConnection.Open(  );

        // создать набор данных и установить его свойство
        myDataSet = new System.Data.DataSet(  );
        myDataSet.CaseSensitive=true;

        // создать объект SqlCommand и присвоить ему
        // объект подключения и объект команды
        myCommand = new System.Data.SqlClient.SqlCommand(  );
        myCommand.Connection=myConnection;
        myCommand.CommandText =
          "Select ShipperID, CompanyName from Shippers";

        // создать dataAdapter, передать ему
        // объект SqlCommand и установить представление таблицы
        dataAdapter = new System.Data.SqlClient.SqlDataAdapter(  );
        dataAdapter.SelectCommand= myCommand;
        dataAdapter.TableMappings.Add("Table","Shippers");

        // Сказать объекту dataAdapter заполнить набор данных dataSet
        dataAdapter.Fill(myDataSet);

        // Установить свойства RadioButtonList
        rbl1.RepeatLayout =
            System.Web.UI.WebControls.RepeatLayout.Flow;
        rbl1.DataTextField = "CompanyName";
        rbl1.DataValueField = "ShipperID";

        // установить источник данных и выполнить его привязку
        rbl1.DataSource = myDataSet.Tables["Shippers"].DefaultView;
        rbl1.DataBind( );

        // выбрать первый переключатель
        rbl1.Items[0].Selected = true;
    }
}

#region Web Form Designer generated code
override protected void OnInit(EventArgs e)
{
    //
    // CODEGEN: This call is required by
    // the ASP.NET Web Form Designer.
    //
    InitializeComponent(  );
    base.OnInit(e);
}

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
/** Метод, необходимый для работы конструктора форм.
```

```
            Не изменяйте его в редакторе исходного кода. **/
        private void InitializeComponent(  )
        {
            this.Order.Click += new System.EventHandler(this.Order_Click);
            this.Load += new System.EventHandler(this.Page_Load);

        }
        #endregion

        private void Order_Click(object sender, System.EventArgs e)
        {
            // сформировать сообщение, получив значения
            // от элементов управления
            string msg;
            msg = "Thank you " + txtName.Text +". You chose " ;
            // iterate over the radio buttons
            for (int i = 0;i<rbl1.Items.Count;i++)
            {
                // если он выбран - добавить его в сообщение
                if (rbl1.Items[i].Selected)
                {
                    msg = msg + rbl1.Items[i].Text;
                    lblFeedBack.Text = msg;
                }  // конец оператора if
            }      // конец цикла for
        }          // конец Order_Click
    }              // конец класса WebForm1
}                  // конец пространства имен ProgrammingCSharpWeb
```

Реакция на отправляющие события

Объекты `<asp:button>` при щелчке автоматически отправляют сообщение формы на сервер. Для обработки этого события не нужно писать специальную программу, если, конечно, не требуется производить никаких дополнительных действий. Если на сервере ничего не предпринять, страница будет просто возвращена клиенту.

Как правило, когда страница снова выводится на экран, каждый элемент управления выводится заново. В силу своей природы Сеть не сохраняет состояния, и если программист хочет сохранить состояние элемента управления (например, восстановить текст, введенный пользователем), он должен делать это вручную. Классическая технология ASP требует, чтобы программист самостоятельно сохранял и восстанавливал состояния элементов, но ASP.NET оказывает в этом определенное содействие. Когда страница отправляется на сервер, к ней автоматически добавляется скрытый элемент по имени `ViewState`:

```
<input type="hidden" name="__VIEWSTATE"
value="YTB6LTI5MTE3ODE1N19hMHpfaHo1ejF4X2Ewel9oejV6NXhfYTB6YTB6YTB6aHpSZXBlY
XRMYXlvdXRfU3lzdGVtLldlYi5VSS5XZWJDb25Ocm9scy5SZXBlYXRMYXlvdXR6VGFibGV4XORhd
```

GFWYWx1ZUZpZWxkX1NoaXBwZXJJRF9EYXRhVGVceHRGaWVsZF9Db21wYW55TmFtZXhfX3hfYTB6Y
TB6YXpTcGVlZHkgRVx4cHJlc3NfMV94X2F6VW5pdGVkIFBhY2thZ2VfMl94X2F6RmVkZXJhbCBTa
GlwcGluZ18zX3hfeF94X3hfX3h4X3h4X3hfX3hcd DUwX1N5c3RlbS5TdHJpbmc=a15204ed" />

Этот элемент управления представляет состояние формы (значения, выбранные пользователем). Когда страница будет выводиться на экран клиента в следующий раз, ASP.NET воспользуется элементом ViewState для установки элементов в их предыдущее состояние.

Когда пользователь щелкает по кнопке Order, страница отправляется на сервер и вызывается обработчик события, связанный с этой кнопкой:

```
public void Order_Click (object sender, System.EventArgs e)
{
    string msg;
    msg = "Thank you " + txtName.Text +". You chose " ;
    for (int i = 0;i<rbl1.Items.Count;i++)
    {
        if (rbl1.Items[i].Selected)
        {
            msg = msg + rbl1.Items[i].Text;
            lblFeedBack.Text = msg;
        }
    }
}
```

Простейшим способом создания обработчика события является двойной щелчок по кнопке Order в режиме визуального проектирования среды Visual Studio .NET. Событие будет добавлено к методу InitializeComponent():

```
Order.Click += new System.EventHandler (this.Order_Click);
```

и будет создана заготовка обработчика события Order_Click. Впрочем, обработчик можно создать и вручную.

На основе введенного имени и выбранного грузоперевозчика этот метод формирует сообщение, которое присваивается метке. Когда форма впервые появляется на экране, она выглядит, как показано на рис. 15.5. Если ввести имя, выбрать, например, United Parcel Service и нажать кнопку Order, форма будет отослана на сервер и снова выведена на экран. Результат этой операции изображен на рис. 15.6.

Форма автоматически запоминает состояния переключателей и текстовых полей (именно для этого предусмотрен элемент ViewState), а на сервере вызывается обработчик события, обновляющий значение метки.

Вниманию программистов, работающих в среде ASP! Ни в *.aspx*-файле, ни в *.cx*-файле нет кода, управляющего состоянием. Программист нигде не сохраняет состояние переключателей и текстовых полей; ASP.NET все делает автоматически.

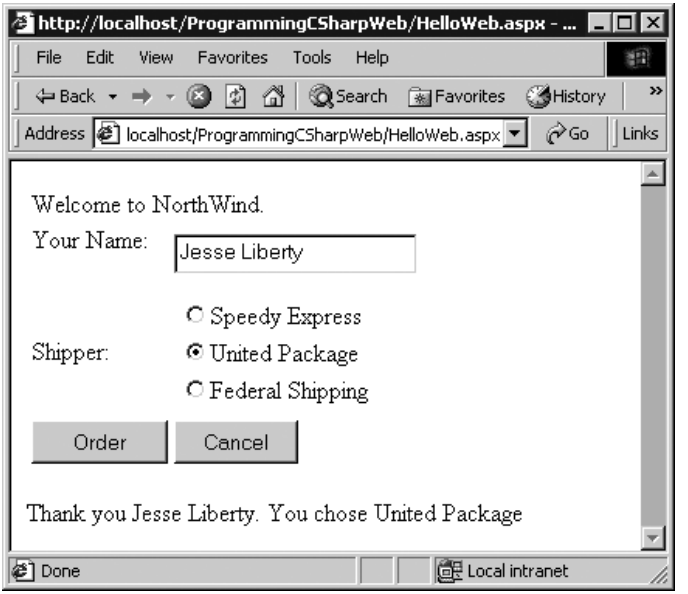

Рис. 15.6. *Страница, отсылаемая после нажатия кнопки Order*

Технология ASP.NET и язык C#

Технологию ASP.NET можно обсуждать довольно долго, но значительная ее часть независима от языка программирования. ASP.NET предоставляет программисту богатый набор элементов управления и других инструментов, в том числе для проверки допустимости данных, представления даты и времени, размещения рекламных объявлений, взаимодействия с пользователем и т. д. Однако эти элементы практически не требуют никакого программирования.

Роль программиста, пишущего на языке C#, сводится к созданию обработчиков событий, которые реагируют на действия пользователя. Многие из таких обработчиков просто участвуют в обмене информацией между базой данных и элементами управления.

16

Веб-службы

Веб-службы платформы .NET (.NET Web Services) расширяют концепцию распределенной обработки, позволяя строить компоненты, методы которых могут быть вызваны через Интернет. Эти компоненты могут быть написаны на любом языке платформы .NET, а связь с ними осуществляется с помощью открытых протоколов, не зависящих от платформы.

Например, сервер биржи может иметь веб-службу в виде метода, который принимает в качестве параметра сокращенное название выпуска акций и возвращает котировку. Приложение может комбинировать эту службу с другой (возможно, предоставляемой третьей стороной), которая тоже принимает название выпуска акций и возвращает подробную информацию об эмитенте (фирме, выпустившей акции). Таким образом, разработчик может сосредоточиться на передаче параметров этим веб-службам, а не на создании дублирующих служб в своем приложении.

Список веб-служб, полезных как для разработчиков, так и для пользователей, кажется бесконечным. Например, книжный магазин может иметь веб-службу, которая по номеру ISBN сообщает, имеется ли книга в наличии и какова ее стоимость. Веб-служба отеля может принимать информацию о количестве туристов и датах заезда, а возвращать подтверждение на бронирование номеров. Можно представить себе веб-службу, которая по телефонному номеру возвращает имя и адрес, или службу, предоставляющую информацию о погоде или о запуске космических кораблей.

Фирма Microsoft анонсировала целый ряд коммерческих .NET-служб как часть своей инициативы .NET My Services. Среди них можно назвать службу Passport для идентификации и удостоверения личности

пользователей (посетите *http://www.passport.com*), а также службы для управления хранением данных, оповещения пользователей, организации совещаний и многие другие. Эти службы, как и службы, создаваемые читателем, можно интегрировать в приложения подобно любым другим коммерческим объектам.

В современном мире одно приложение может объединять в себе услуги сотен маленьких веб-служб, разбросанных по всему свету. Это переводит Сеть на качественно новый уровень, позволяя не только получать информацию и обмениваться ею, но и вызывать методы и исполнять приложения.

SOAP, WSDL и Discovery

Что действительно необходимо веб-службам, так это простой, повсеместно принятый протокол, позволяющий обращаться к веб-службам, а также находить и вызывать сервисы других служб. В 1999 году консорциуму World Wide Web Consortium был предложен на рассмотрение протокол SOAP (Simple Object Access Protocol, простой протокол доступа к объектам). SOAP имеет то достоинство, что он основан на языке XML и использует стандартные интернет-протоколы передачи данных.

SOAP – это упрощенный протокол передачи сообщений, построенный на основе XML, HTTP и SMTP. Чтобы клиент мог взаимодействовать с веб-службой через протокол SOAP, желательно, но необязательно наличие еще двух протоколов: понятного клиентам описания методов, предоставляемых конкретной веб-службой, и описания всех веб-служб, расположенных на этом сайте или по этому URL-адресу. Первое описание обеспечивается в .NET протоколом WSDL (Web Service Description Language, язык описания веб-службы), совместно разработанным такими фирмами, как Microsoft, IBM и некоторыми другими. Для осуществления поиска предложены еще два протокола: UDDI, плод сотрудничества нескольких фирм, включая Microsoft и IBM, а также Discovery, эксклюзивное предложение от Microsoft.

WSDL является XML-схемой, применяемой для описания методов, предоставляемых веб-службой (то есть ее интерфейса). Discovery позволяет приложениям обнаруживать и запрашивать описания веб-служб, что является предварительным этапом перед обращением к веб-службе. Именно на этом этапе будущие пользователи веб-службы узнают о ее существовании, возможностях и о способах взаимодействия с ней. Файл Discovery (с расширением *.disco*) предоставляет броузерам информацию, позволяющую при посещении веб-сайта легко определить URL-адреса его веб-служб. Когда сервер получает запрос на *.disco*-файл, он создает список некоторых или всех URL-адресов на сайте, по которым доступны веб-службы.

Поддержка на стороне сервера

Информация, необходимая для обнаружения и вызова веб-службы, интегрирована в платформу .NET Framework и предоставляется классами из пространства имен `System.Web.Services`. Создание веб-службы не требует от программиста каких-то особых действий. Достаточно написать код реализации, добавить атрибут `[WebMethod]` и позволить серверу сделать все остальное. Атрибуты подробно обсуждаются в главе 18.

Поддержка на стороне клиента

Для обращения к сервисам веб-службы используются клиентские программы, работающие так, словно обращаются непосредственно к хост-серверу через URL-адрес. Однако в действительности клиент взаимодействует с *посредником* (*proxy*). Задачей посредника является представление сервера на компьютере клиента, то есть объединение запросов клиента в сообщения протокола SOAP, которые посылаются серверу, и получение от сервера ответов на запросы. Посредники и работа с объектами на другом компьютере подробно обсуждаются в главе 19.

Построение веб-службы

Чтобы продемонстрировать технологию создания веб-службы на языке C# с помощью классов платформы .NET Framework, создадим простую программу-калькулятор и сделаем ее функции доступными через Сеть.

Начнем со специфицирования веб-службы. Для этого определим класс, производный от `System.Web.Services.WebService`. Простейший способ сделать это – открыть Visual Studio .NET и создать новый проект C# Web Service. По умолчанию среда Visual Studio .NET назовет его `WebService1`, но программист может выбрать более подходящее имя.

Visual Studio .NET создает заготовку веб-службы и даже предоставляет демонстрационный метод, который можно заменить на собственный (пример 16.1).

Пример 16.1. Заготовка класса, созданная средой Visual Studio .NET

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WSCalc
{
    /// <summary>
    /// Summary description for Service1.
    /// </summary>
```

```
    public class Service1 : System.Web.Services.WebService
    {
        public Service1(  )
        {
            //CODEGEN: This call is required by
            // the ASP.NET Web Services Designer

            InitializeComponent(  );
        }

        #region Component Designer generated code

        //Required by the Web Services Designer
                private IContainer components = null;

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>

        private void InitializeComponent(  )
        {
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if(disposing && components != null)
            {
                components.Dispose(  );
            }
            base.Dispose(disposing);
        }

        #endregion

        // WEB SERVICE EXAMPLE
        // The HelloWorld(  ) example service
        // returns the string Hello World
        // To build, uncomment the following lines
        // then save and build the project
        // To test this web service, press F5


//        [WebMethod]
//        public string HelloWorld(  )
//        {
//            return "Hello World";
//        }
    }
}
```

Для построения калькулятора понадобятся пять методов: `Add()`, `Sub()`, `Mult()`, `Div()` и `Pow()`. Каждый из них имеет два формальных параметра

типа double, выполняет необходимое действие и возвращает значение того же типа. Вот, например, метод возведения числа в указанную степень[1]:

```
public double Pow(double x, double y)
{
   double retVal = x;
   for (int i = 0;i < y-1;i++)
   {
      retVal *= x;
   }
   return retVal;
}
```

Чтобы представить метод в виде веб-службы, перед его объявлением необходимо поставить атрибут [WebMethod] (атрибуты обсуждаются в главе 18):

```
[WebMethod]
```

Вовсе не обязательно представлять все методы класса в виде веб-служб. Разработчик вправе решать, какие методы сопровождать атрибутом [WebMethod], а какие – нет.

Это все, что требуется от программиста; платформа .NET сделает все остальное.

В примере 16.2 приведен полный исходный текст приложения веб-службы Калькулятор.

Пример 16.2. Веб-служба Калькулятор

```
using System;
using System.Collections;
```

WSDL и пространства имен

Веб-служба использует Web Service Description Language XML-документ, описывающий доступные через Сеть точки вызова. В каждом таком документе должно быть указано пространство имен XML для гарантии того, что все методы имеют уникальные имена. Пространством имен по умолчанию является *http://tempuri.org/*, но разработчики обычно меняют его перед публикацией своих веб-служб.

Имя пространства имен XML может быть задано в атрибуте WebService:

```
[WebService(Namespace="http://www.LibertyAssociates.com/webServices/")]
```

Более подробно атрибуты описаны в главе 18.

[1] Явно это автором не оговаривается, однако подразумевается, что число возводится в целую положительную степень. – *Примеч. науч. ред.*

```
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WSCalc
{
    [WebService(Namespace="http://www.libertyAssociates.com/webServices/")]
    public class Service1 : System.Web.Services.WebService
    {
        public Service1(  )
        {
            //CODEGEN: This call is required by the
            //ASP.NET Web Services Designer

            InitializeComponent(  );
        }

        #region Component Designer generated code

        //Required by the Web Services Designer
                private IContainer components = null;

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>

        private void InitializeComponent(  )
        {
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>

        protected override void Dispose( bool disposing )
        {
            if(disposing && components != null)
            {
                components.Dispose(  );
            }
            base.Dispose(disposing);
        }

        #endregion

        [WebMethod]
        public double Add(double x, double y)
        {
            return x+y;
        }

        [WebMethod]
        public double Sub(double x, double y)
        {
            return x-y;
```

```
        }
        [WebMethod]
        public double Mult(double x, double y)
        {
            return x*y;
        }
        [WebMethod]
        public double Div(double x, double y)
        {
            return x/y;
        }
        [WebMethod]
        public double Pow(double x, double y)
        {
            double retVal = x;
            for (int i = 0;i < y-1;i++)
            {
                retVal *= x;
            }
            return retVal;
        }
    }
}
```

При компиляции этого проекта в среде Visual Studio .NET создается библиотека DLL, которая помещается в соответствующий подкаталог Интернет-сервера (например в *c:\InetPub\wwwroot\WSCalc*). Взглянув на содержимое этого каталога, нетрудно убедиться, что *.disco*-файл там тоже присутствует.

Создавать сервер в среде Visual Studio .NET совсем не обязательно, при желании можно обойтись редактором Notepad. Просто Visual Studio .NET избавляет программиста от рутинной работы по созданию каталогов, *.disco*-файла и т. д. Особенно удобна среда Visual Studio .NET при создании клиентских файлов, что будет вскоре продемонстрировано.

Тестирование веб-службы

Если указать в броузере URL-адрес созданной веб-службы (или открыть броузер, запустив выполнение программы в среде Visual Studio .NET), появится автоматически созданная серверная веб-страница, описывающая веб-службу (рис. 16.1). Подобные страницы являются хорошим средством тестирования веб-служб. (В следующем разделе поясняется, откуда берутся такие тестовые страницы.)

Если щелкнуть по ссылке с именем какого-либо метода, появится страница с его описанием. Отсюда можно вызвать метод, введя значения его аргументов и нажав кнопку Invoke (рис. 16.2).

Если в первое поле ввести значение `38`, а во второе `4`, это будет запросом к веб-службе на возведение числа 38 в четвертую степень. Результатом

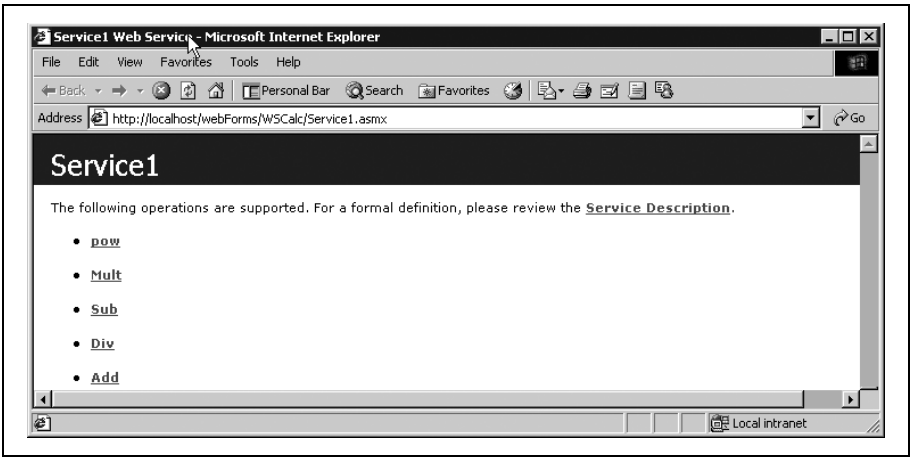

Рис. 16.1. Страница с веб-службой

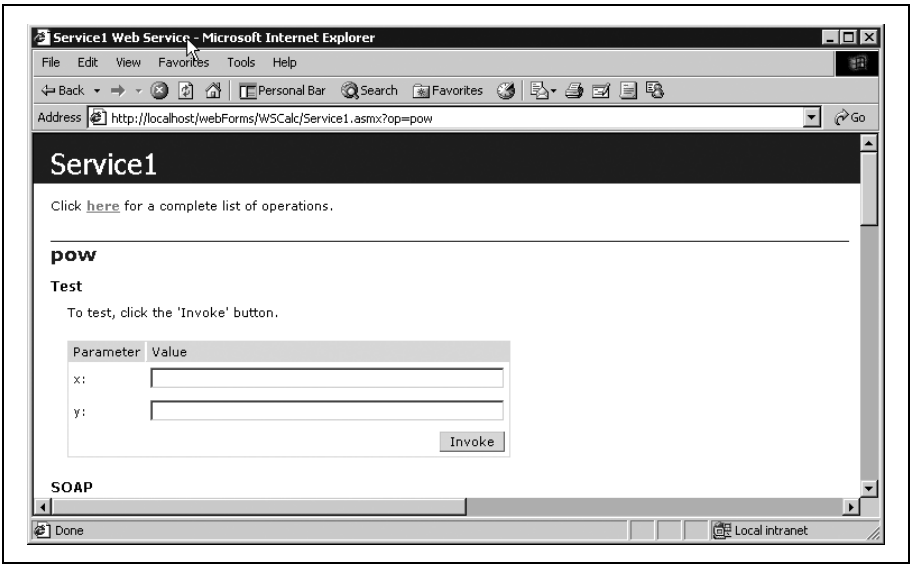

Рис. 16.2. Тестовая страница для метода веб-службы

является XML-страница, описывающая возвращаемые данные. Она изображена на рис. 16.3.

Обратите внимание, что значения аргументов 38 и 4 закодированы в URL, а выходной XML-код содержит результат возведения 38 в степень 4, равный 2 085 136.

Просмотр контракта WSDL

Значительная часть работы по созданию веб-службы выполняется автоматически. Среда создает HTML-страницы, описывающие веб-служ-

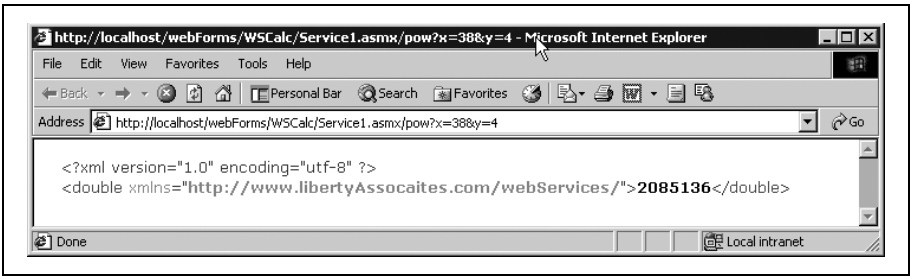

Рис. 16.3. *XML-код с результатом вызова метода*

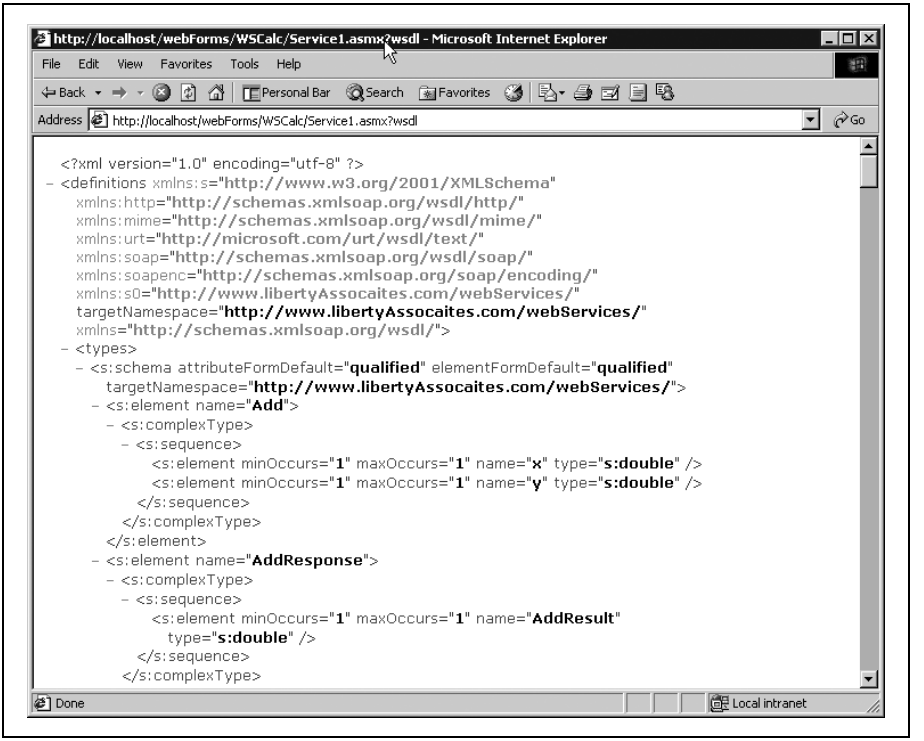

Рис. 16.4. *WSDL-документ веб-службы Калькулятор*

бу и ее методы, а эти страницы содержат ссылки на страницы, которые позволяют протестировать методы веб-службы. Как это делается?

Выше уже говорилось, что веб-служба описывается в WSDL. Чтобы просмотреть WSDL-документ, следует добавить `?WSDL` к URL-адресу веб-службы:

```
http://localhost/WSCalc/Service1.asmx?wsdl
```

Броузер выведет документ на экран, как показано на рис. 16.4.

Подробный разбор содержимого WSDL-документа выходит за рамки этой книги, однако читатель может заметить, что каждый метод пол-

ностью описан в структурированном XML-формате. Эта информация используется протоколом SOAP и позволяет броузеру клиента вызывать методы веб-службы на стороне сервера.

Создание класса-посредника

Прежде чем создавать клиентское приложение, взаимодействующее с веб-службой Калькулятор, необходимо создать класс-посредник. Конечно, его можно написать вручную, но это тяжелая и утомительная работа. Программисты из фирмы Microsoft представили разработчикам инструментальное средство под названием `wsdl`, которое создает исходный текст класса-посредника на основании информации из WSDL-файла.

Чтобы создать класс-посредник, следует в командной строке ввести команду `wsdl`, а за ней – путь к WSDL-документу. Например:

```
wsdl http://localhost/WSCalc/Service1.asmx?wsdl
```

Результатом будет создание клиентского файла на языке C# с именем *Service1.cs*, фрагмент из которого представлен в примере 16.3. Вы должны добавить имя пространства имен `WSCalc`, поскольку оно понадобится при построении класса, а команда `wsdl` его не добавляет.

Пример 16.3. Примерный текст программы клиента, необходимый для обращения к веб-службе Калькулятор

```
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.Web.Services;

namespace WSCalc
{
   [System.Web.Services.WebServiceBindingAttribute(
       Name="Service1Soap",
       Namespace="http://www.libertyAssociates.com/webServices/")]
   public class Service1 :
      System.Web.Services.Protocols.SoapHttpClientProtocol
   {

      public Service1()
      {
         this.Url = "http://localhost/WSCalc/service1.asmx";
      }

      [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
          "http://www.libertyAssociates.com/webServices/Add",
          RequestNamespace=
            "http://www.libertyAssociates.com/webServices/",
          ResponseNamespace=
             "http://www.libertyAssociates.com/webServices/",
          Use=System.Web.Services.Description.SoapBindingUse.Literal,
```

```
            ParameterStyle=
               System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
         public System.Double Add(System.Double x, System.Double y)
         {
            object[] results =
               this.Invoke("Add", new object[] {x,y});
            return ((System.Double)(results[0]));
         }

      public System.IAsyncResult
            BeginAdd(System.Double x, System.Double y,
            System.AsyncCallback callback, object asyncState)
         {
            return this.BeginInvoke("Add", new object[] {x,
                        y}, callback, asyncState);
         }

         public System.Double EndAdd(System.IAsyncResult asyncResult)
         {
            object[] results = this.EndInvoke(asyncResult);
            return ((System.Double)(results[0]));
         }
```

Эта довольно сложная программа создается утилитой WSDL и применяется для построения библиотеки DLL класса-посредника. Она необходима при построении клиентского приложения. В этом файле широко используются атрибуты (см. главу 18), но читатель уже знает достаточно, чтобы догадаться о предназначении хотя бы некоторых из них.

Файл начинается с объявления класса `Service1`, являющегося потомком класса `SoapHttpClientProtocol`, принадлежащего пространству имен `System.Web.Services.Protocols`:

```
public class Service1 : System.Web.Services.Protocols.SoapHttpClientProtocol
```

Конструктор этого класса устанавливает свойство URL, наследуемое от класса `SoapHttpClientProtocol`, равным URL-адресу страницы *.asmx*, созданной ранее.

Метод `Add()` объявляется с атрибутами хоста, которые заставляют SOAP выполнить работу, связанную с удаленным вызовом метода.

Кроме того, приложение WSDL предоставляет методам асинхронную поддержку. Например, для метода `Add()` оно создает методы `BeginAdd()` и `EndAdd()`, которые позволяют взаимодействовать с веб-службой без потери производительности.

Для того чтобы построить класс-посредник, программист должен добавить сгенерированный утилитой WSDL код в проект C# Library среды Visual Studio .NET. Затем следует скомпилировать этот проект, создающий библиотеку DLL. Обязательно запишите местоположение этой библиотеки, так как она понадобится при компиляции клиентского приложения.

Для тестирования веб-службы напишем очень простое консольное приложение на языке C#. Единственным нетривиальным моментом будет добавление ссылки на только что созданную библиотеку DLL. Когда это будет сделано, можно создавать объект веб-службы, как создается любой другой локальный объект:

```
WSCalc.Service1 theWebSvc = new WSCalc.Service1();
```

Теперь можно вызвать метод Pow() так, словно это метод локального объекта:

```
for (int i = 2;i<10; i++)
    for (int j = 1;j <10;j++)
    {
       Console.WriteLine(
         "{0} в степени {1} = {2}", i, j,
        theWebSvc.Pow(i, j));
    }
```

В этом цикле создается таблица возведения в степень от 1 до 9 чисел от 2 до 9. Полный исходный текст приложения и фрагмент выводимой им информации содержатся в примере 16.4.

Пример 16.4. Программа клиента для тестирования веб-службы Калькулятор

```
using System;

// программа для тестирования веб-службы
public class Tester
{
   public static void Main()
   {
      Tester t = new Tester();
      t.Run();
   }

   public void Run()
   {
      int var1 = 5;
      int var2 = 7;

      // создание экземпляра класса заместителя веб-службы
      WSCalc.Service1 theWebSvc = new WSCalc.Service1();

      // вызов метода Add()

      Console.WriteLine("{0} + {1} = {2}", var1, var2,
         theWebSvc.Add(var1, var2));

      // построение таблицы путем многократного вызова метода pow()
      for (int i = 2;i<10; i++)
         for (int j = 1;j <10;j++)
         {
```

```
            Console.WriteLine("{0} в степени {1} = {2}", i, j,
               theWebSvc.Pow(i, j));
         }
   }
}
```

Вывод (отрывок):

```
5 + 7 = 12
2 в степени 1 = 2
2 в степени 2 = 4
2 в степени 3 = 8
2 в степени 4 = 16
2 в степени 5 = 32
2 в степени 6 = 64
2 в степени 7 = 128
2 в степени 8 = 256
2 в степени 9 = 512
3 в степени 1 = 3
3 в степени 2 = 9
3 в степени 3 = 27
3 в степени 4 = 81
3 в степени 5 = 243
3 в степени 6 = 729
3 в степени 7 = 2187
3 в степени 8 = 6561
3 в степени 9 = 19683
```

Теперь эта служба даже более доступна в Сети, чем читатель мог предположить еще совсем недавно (конечно, с поправкой на параметры безопасности). К ней можно обратиться с помощью протоколов HTTP-GET, HTTP-POST и SOAP. В созданном клиентском приложении использован протокол SOAP, но ничто не мешает создать приложение, передающее запросы по протоколу HTTP-GET:

```
http://localhost/WSCalc/WebService1.asmx/Add?x=23&y=22
```

Если ввести этот URL в адресное окно броузера, он ответит так:

```
<?xml version="1.0" encoding="utf-8"?>
  <double xmlns="http://www.libertyAssociates.com/
                 webServices/">45</double>
```

Основным преимуществом протокола SOAP над HTTP-GET и HTTP-POST является поддержка им широкого спектра типов данных, включая базовые типы C# (`int`, `double` и т. д.), а также перечислимых типов, классов, структур, наборов данных DataSet ADO.NET и массивов элементов любого из этих типов.

Кроме того, в то время как протоколы HTTP-GET и HTTP-POST ограничены парами «имя/значение» базовых типов и перечислений, богатый, основанный на XML синтаксис протокола SOAP предлагает более устойчивый альтернативный способ обмена данными.

III

CLR и .NET Framework

17

Сборки и контроль версий

Базовой единицей программирования на платформе .NET является *сборка* (*assembly*). Сборка – это коллекция файлов, которая предстает перед пользователем в виде библиотеки динамической компоновки (DLL) или исполняемого файла (EXE). Библиотеки DLL являются коллекциями классов и методов, которые связываются с работающей программой только в случае необходимости.

Сборки являются базовыми единицами платформы .NET для повторного использования кода, контроля версий, защиты и развертывания. В этой главе сборки обсуждаются достаточно подробно, в частности рассматриваются их архитектура и содержимое, а также закрытые и совместно используемые сборки.

Помимо объектного кода приложения сборки содержат ресурсы (например *.gif*-файлы), определения типов для каждого класса, созданного разработчиком, а также метаданные, то есть информацию о коде и данных. Метаданные подробно изучаются в главе 18.

PE-файлы

Сборки хранятся на диске в виде переносимых исполняемых (Portable Executable) файлов, или PE-файлов. Концепция PE-файлов не нова. Формат PE-файлов платформы .NET точно такой же, как в Windows, а реализуются они в виде библиотек DLL или EXE-файлов. Логически (в смысле «не физически») сборки состоят из одного или нескольких *модулей*. Однако необходимо заметить, что сборка должна иметь только одну точку входа: `DLLMain`, `WinMain` или `Main`. При этом `DLLMain` является точкой входа в DLL, `WinMain` – точкой входа в приложение Windows, а `Main` – в консольное или DOS-приложение.

Модули создаются в виде библиотек DLL и являются составными частями сборки. Сами по себе они не могут быть выполнены и, чтобы их можно было использовать, должны объединяться в сборки.

Распространяться и повторно использоваться может только вся сборка целиком. Сборки загружаются по требованию и не загружаются, если они не используются.

Метаданные

Метаданные (*etadata*) – это сохраняемая в сборке информация, описывающая типы и методы сборки, а также предоставляющая о ней некоторые другие сведения. О сборке можно сказать, что она *самодокументируемая*, поскольку метаданные полностью описывают содержимое каждого модуля. Более подробно метаданные обсуждаются в главе 18.

Границы безопасности

Сборки образуют границы безопасности, а также границы типов. Иными словами, сборка является областью видимости типов, которые она содержит, и типам нельзя пересекать ее границы. Конечно, программист может ссылаться на типы из программы, расположенной за границами сборки, но для этого нужно создать специальную ссылку на эту сборку либо в интегрированной среде разработки, либо в командной строке на этапе компиляции. Чего действительно нельзя делать, так это «растягивать» определение типа на две сборки.

Контроль версий

Каждая сборка имеет номер версии, и версии не могут пересекать границы сборки. Иными словами, версия может относиться только к содержимому одной сборки. Все типы и ресурсы внутри сборки меняют свои версии одновременно.

Манифесты

Составной частью метаданных каждой сборки является *манифест* (*manifest*), описывающий содержимое сборки, включая идентификационную информацию (имя, версию и т. д.), список типов и ресурсов, карта отображения открытых типов на реализующий код и список сборок, на которые ссылается данная сборка.

Манифест есть даже у самой простой программы. Его можно просмотреть с помощью утилиты `ILDasm`, являющейся составной частью среды разработки. Если открыть исполняемую программу, приведенную в

примере 12.3, с помощью ILDasm, то на экране появится диалоговое окно, изображенное на рис. 17.1.

Рис. 17.1. *Окно утилиты ILDasm для примера 12.3*

Обратите внимание на манифест (вторая строчка сверху). Двойной щелчок левой кнопкой мыши по нему откроет окно Manifest (рис. 17.2).

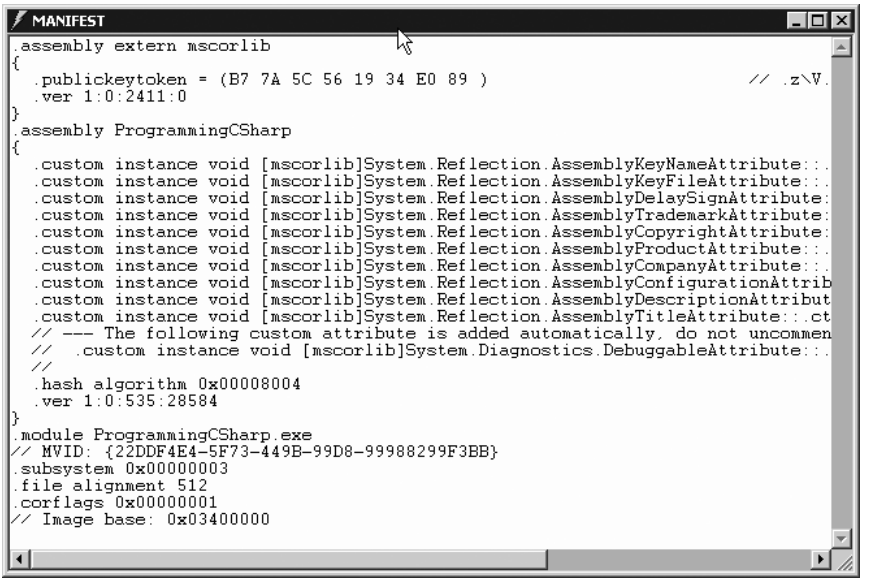

Рис. 17.2. *Окно Manifest*

Этот файл является картой содержимого сборки. В первой строке расположена ссылка на сборку `mscorlib`, на которую ссылается данная сборка, равно как и любое приложение платформы .NET. Сборка `mscorlib` представляет собой библиотеку ядра .NET; она имеется на каждой платформе .NET.

В следующей строчке манифеста сборки можно видеть ссылку на сборку программы из примера 12.3, причем эта сборка состоит из единственного модуля. Остальными метаданными пока можно пренебречь.

Модули в манифесте

Сборка может состоять из нескольких модулей. В этом случае манифест включает в себя хеш-код, идентифицирующий каждый модуль для гарантии того, что при выполнении программы будет загружена именно та версия модуля, которая требуется. Если у какого-либо модуля есть несколько версий, хеш-код обеспечит корректную загрузку программы.

Хеш-код – это числовое представление кода модуля. Если код модуля изменится, прежний хеш-код не будет ему соответствовать.

Манифесты модулей

У каждого модуля есть собственный манифест, существующий отдельно от манифеста сборки. Манифест модуля перечисляет сборки, на которые этот модуль ссылается. Кроме того, модуль может содержать ресурсы (например, изображения), которые нужны ему для работы.

Другие сборки

Манифест сборки содержит, кроме прочего, ссылки на другие сборки, необходимые данной. Каждая такая ссылка включает в себя имя требуемой сборки, номер версии и необходимые региональные настройки (culture), а также автора сборки (последнее, впрочем, необязательно). Автор (разработчик или фирма) представлен цифровой подписью (уникальным идентификатором).

Региональные настройки – это объект, включающий в себя используемый язык и некоторые характеристики вывода информации на экран. Он должен соответствовать ожиданиям пользователя программы. Например, именно этот объект определяет формат вывода даты: месяц/день/год или день/месяц/год.

Многомодульные сборки

Одномодульная сборка представляет собой один-единственный файл, EXE или DLL. Этот модуль содержит все типы и код реализации приложения, причем этот же файл содержит и манифест сборки.

Многомодульная сборка состоит из нескольких файлов (не более одного исполняемого файла и любое число файлов библиотек, причем сборка обязательно должна содержать хотя бы один исполняемый файл или файл библиотеки динамической компоновки). В этом случае манифест сборки может находиться в отдельном файле, но может быть помещен и в один из модулей. Когда программа вызовет сборку, среда CLR загрузит файл с манифестом, а затем загрузит требуемые модули по мере необходимости.

Достоинства многомодульных сборок

В реальных приложениях следует отдавать предпочтение многомодульным сборкам, особенно если они разрабатываются несколькими программистами или имеют большой размер.

Представим, что 25 человек работают над одним проектом. Если бы они создавали одномодульную сборку, то при построении и тестировании приложения всем 25 программистам пришлось бы проверять свою программу одновременно, а сопровождение такого приложения-монстра превратилось бы в сплошной кошмар.

Зато если каждый будет создавать свой модуль, при компоновке программы можно брать самые последние версии модулей. Это снимает проблемы сопровождения, поскольку модули можно проверять по мере их готовности.

Пожалуй, более важным достоинством многомодульности является простота распространения больших программ. Если каждый из 25 программистов построит отдельный модуль в отдельном DLL-файле, то сотрудник, ответственный за построение приложения, создаст 26-й модуль с манифестом сборки. Эти 26 файлов можно легко перенести на компьютер конечного пользователя. Тому будет достаточно загрузить один модуль с манифестом, не беспокоясь об остальных 25. Манифест сам разберется, где какой метод, и будет загружать модули по мере вызова методов. Такая схема вполне прозрачна для пользователя.

При появлении новых версий программистам будет достаточно отослать пользователю обновленные модули (и обновленный манифест). Можно добавлять новые модули или отказываться от уже существующих, а конечный пользователь будет по-прежнему загружать только один модуль с манифестом.

Кроме прочего, весьма вероятно, что не все 25 модулей понадобятся программе одновременно. Если программа разбита на 25 модулей, будут загружаться только действительно необходимые части программы. Редко используемый код можно будет выделить в отдельный модуль, который вообще не загружается при нормальном развитии событий. Это все была теория. Что касается практики, платформе .NET удается обойтись без кошмаров, традиционно сопровождающих DLL, и это является грандиозным достижением. «Ад DLL» (DLL Hell) обсуждается далее в этой главе.

Построение многомодульной сборки

Для демонстрации многомодульной сборки создадим в следующем примере пару очень простых модулей и объединим их в одну сборку. Первым модулем будет класс Fraction. Этот несложный класс позволяет создавать простые дроби и выполнять действия с ними. Он представлен в примере 17.1.

Пример 17.1. Класс Fraction

```
namespace ProgCS
{
    using System;

    public class Fraction
    {
        public Fraction(int numerator, int denominator)
        {
            this.numerator = numerator;
            this.denominator = denominator;
        }

        public Fraction Add(Fraction rhs)
        {
            if (rhs.denominator != this.denominator)
            {
                throw new ArgumentException(
                     "Знаменатели должны соответствовать.");
            }

            return new Fraction(
                this.numerator + rhs.numerator,
                     this.denominator);
        }

        public override string ToString()
        {
            return numerator + "/" + denominator;
        }

        private int numerator;
        private int denominator;
    }
}
```

Обратите внимание, что класс Fraction находится в пространстве имен ProgCS. Полное имя класса – ProgCS.Fraction.

Конструктор класса Fraction имеет два формальных параметра, numerator и denominator (числитель и знаменатель соответственно). В класс входит метод Add(), который принимает в качестве аргумента вторую дробь и возвращает сумму (предполагается, что складываемые дроби имеют одинаковые знаменатели). Этот класс упрощен до предела, но он обладает функциональностью, достаточной для данного примера.

Вторым классом будет myCalc, реализующий калькулятор (пример 17.2).

Пример 17.2. Калькулятор

```
namespace ProgCS
{
    using System;

    public class myCalc
    {
        public int Add(int val1, int val2)
        {
            return val1 + val2;
        }
        public int Mult(int val1, int val2)
        {
            return val1 * val2;
        }
    }
}
```

И в этом классе нет ничего «лишнего». Между прочим, он тоже находится в пространстве имен ProgCS.

Этих двух классов достаточно для создания сборки. Файл *AssemblyInfo.cs* будет содержать метаданные этой сборки. (Метаданные обсуждаются в главе 18.)

Конечно, читатель может создать файл *AssemblyInfo.cs* самостоятельно, но гораздо проще создать его средствами Visual Studio .NET.

По умолчанию Visual Studio .NET создает одномодульные сборки. Для создания многомодульного ресурса укажите в командной строке параметр /addModules. Простейший способ откомпилировать и скомпоновать многомодульную сборку состоит в использовании make-файла, который нетрудно создать с помощью любого текстового редактора, например Notepad.

Если вы не знакомы с make-файлами, беспокоиться не стоит. Данный пример – единственный, где требуется make-файл, да и то лишь для преодоления упомянутой установки по умолчанию Visual Studio .NET. В крайнем случае, make-файлом, представленным ниже, можно воспользоваться и не понимая до конца каждую его строчку.

В примере 17.3 приведен законченный make-файл (который тут же подробно разъясняется). Чтобы запустить этот пример, разместите make-файл с именем *makefile* в одном каталоге с копиями *Calc.cs*, *Fraction.cs* и *AssemblyInfo.cs*. Откройте командное окно .NET и перейдите в эту

папку. Запустите *nmake* без каких-либо параметров командной строки. После выполнения этих операций в подкаталоге *\bin* будет создан файл *SharedAssembly.dll*.

Пример 17.3. Make-файл для многомодульной сборки

```
ASSEMBLY= MySharedAssembly.dll

BIN=.\bin
SRC=.
DEST=.\bin

CSC=csc /nologo /debug+ /d:DEBUG /d:TRACE

MODULETARGET=/t:module
LIBTARGET=/t:library
EXETARGET=/t:exe

REFERENCES=System.dll

MODULES=$(DEST)\Fraction.dll $(DEST)\Calc.dll
METADATA=$(SRC)\AssemblyInfo.cs

all: $(DEST)\MySharedAssembly.dll

# Метаданные сборки помещены в тот же модуль, что и манифест
$(DEST)\$(ASSEMBLY): $(METADATA) $(MODULES) $(DEST)
    $(CSC) $(LIBTARGET) /addmodule:$(MODULES: =;) /out:$@ %s

# Добавить в список зависимостей модуль Calc.dll
$(DEST)\Calc.dll: Calc.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES: =;) /out:$@ %s

# Добавить класс Fraction
$(DEST)\Fraction.dll: Fraction.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES: =;) /out:$@ %s

$(DEST)::
!if !EXISTS($(DEST))
        mkdir $(DEST)
!endif
```

Этот файл начинается с определения сборки, которую необходимо построить:

```
ASSEMBLY= MySharedAssembly.dll
```

Затем определяются каталоги. Исходный код берется из текущего каталога, а результат помещается в подкаталог *bin* текущего каталога:

```
BIN=.\bin
SRC=.
DEST=.\bin
```

Сборка строится следующим образом:

```
$(DEST)\$(ASSEMBLY): $(METADATA) $(MODULES) $(DEST)
    $(CSC) $(LIBTARGET) /addmodule:$(MODULES: =;) /out:$@ %s
```

Этот фрагмент файла сообщает программе nmake (выполняющей make-файл), что сборка *MySharedAssembly.dll* будет размещена в каталоге (*bin*) и состоит из метаданных и модулей. Кроме того, здесь указывается командная строка, необходимая для компиляции.

Метаданные определяются чуть выше:

```
METADATA=$(SRC)\AssemblyInfo.cs
```

Модули определяются как две библиотеки динамической компоновки:

```
MODULES=$(DEST)\Fraction.dll $(DEST)\Calc.dll
```

Командная строка компиляции строит библиотеку и добавляет в нее модули, помещая результат в файл сборки *MySharedAssembly.dll*:

```
$(DEST)\$(ASSEMBLY): $(METADATA) $(MODULES) $(DEST)
    $(CSC) $(LIBTARGET) /addmodule:$(MODULES: =;) /out:$@ %s
```

Чтобы выполнить все это, программа nmake должна уметь создавать модули. Вначале следует сообщить ей, как создать *calc.dll*. Для этого требуются исходный файл *calc.cs* и командная строка для построения DLL-библиотеки:

```
$(DEST)\Calc.dll: Calc.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES: =;) /out:$@ %s
```

Аналогично создается *fraction.dll*:

```
$(DEST)\Fraction.dll: Fraction.cs $(DEST)
    $(CSC) $(MODULETARGET) /r:$(REFERENCES: =;) /out:$@ %s
```

Результатом обработки этого make-файла программой nmake будут три библиотеки: *fraction.dll*, *calc.dll* и *MySharedAssembly.dll*. Если открыть *MySharedAssembly.dll* с помощью ILDasm, то окажется, что она состоит лишь из манифеста, что показано на рис. 17.3.

Рис. 17.3. *Сборка MySharedAssembly.dll*

В этом манифесте нетрудно обнаружить метаданные для только что созданных библиотек (рис. 17.4).

```
MANIFEST
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                          // .z\V.4..
  .hash = (8B BB 5A BD 8D A3 12 7D 08 A2 25 D0 48 17 28 4F   // ..Z....}..%.H.(O
          20 57 EA 07 )                                      //  W..
  .ver 1:0:2411:0
}
.assembly MySharedAssembly
{
  // --- The following custom attribute is added automatically, do not uncomment -------
  //   .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(bool,
  //                                                                                 bool) = ( 01 00 01
  .custom instance void [mscorlib]System.Reflection.AssemblyKeyNameAttribute::.ctor(string) = ( 01 00
  .custom instance void [mscorlib]System.Reflection.AssemblyKeyFileAttribute::.ctor(string) = ( 01 00
  .custom instance void [mscorlib]System.Reflection.AssemblyDelaySignAttribute::.ctor(bool) = ( 01 00
  .custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string) = ( 01 0
  .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = ( 01 0
  .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) = ( 01 00
  .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) = ( 01 00
  .custom instance void [mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = (
  .custom instance void [mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = ( 01 00 00
  .hash algorithm 0x00008004
  .ver 1:0:535:29053
}
.file Fraction.dll
    .hash = (DB EB 60 2F E3 2C F3 6B 55 3F 34 0A 50 EE CD 51   // ..`/.,.kU?4.P..Q
            F4 44 58 C6 )                                      // .DX.
.file Calc.dll
    .hash = (92 1C 3D AB 27 E2 0D FE 60 62 94 9E 6C 4C 63 03   // ..=.'...`b..lLc.
            5C 4F 1D CD )                                      // \O..
.class extern public ProgCS.Fraction
{
  .file Fraction.dll
  .class 0x02000002
}
.class extern public ProgCS.myCalc
{
  .file Calc.dll
  .class 0x02000002
}
.module MySharedAssembly.dll
// MVID: {F5D8C740-E1DA-465A-A528-B861CA6E04E8}
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03440000
```

Рис. 17.4. *Манифест сборки MySharedAssembly.dll*

Вначале идет внешняя сборка для библиотеки ядра `mscorlib`, а за ней – два модуля, `ProgCS.Fraction` и `ProgCS.myCalc`. Итак, налицо сборка, состоящая из трех DLL-файлов: *MySharedAssembly.dll* с манифестом, а также *Calc.dll* и *Fraction.dll* с необходимыми типами и исполняемым кодом.

Тестирование сборки

Чтобы воспользоваться этой сборкой, нужно создать тестовую программу, которая будет загружать модули по мере необходимости. В примере 17.4 такая программа приведена. Сохраните ее в файле *Test.cs* в том же каталоге, что и остальные модули.

Пример 17.4. Тестирующая программа

```
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        // метод Main() не загрузит совместно используемую сборку
        static void Main()
        {
```

```
            Test t = new Test();
            t.UseCS();
            t.UseFraction();

        }

        // Вызов этого метода загрузит сборки myCalc и mySharedAssembly
        public void UseCS()
        {
            ProgCS.myCalc calc = new ProgCS.myCalc();
            Console.WriteLine("3+5 = {0}\n3*5 = {1}",
                      calc.Add(3,5), calc.Mult(3,5));
        }

        // Вызов этого метода загрузит сборку Fraction
        public void UseFraction()
        {
            ProgCS.Fraction frac1 = new ProgCS.Fraction(3,5);
            ProgCS.Fraction frac2 = new ProgCS.Fraction(1,5);
            ProgCS.Fraction frac3 = frac1.Add(frac2);
            Console.WriteLine("{0} + {1} = {2}", frac1, frac2, frac3);
        }
    }
}
```

Вывод:

```
3+5 = 8
3*5 = 15
3/5 + 1/5 = 4/5
```

В методе `Main()` отсутствует какой-либо программный текст, связанный с вашими модулями. Это сделано для демонстрации их загрузки. Поскольку при загрузке метода `Main()` не требуется, чтобы загружались другие модули, объектов `Fraction` и `Calc` в нем нет. Зато когда вызываются методы `UseFraction` и `UseCalc`, можно наблюдать, как загружаются требуемые модули.

Загрузка сборки

Сборка загружается в приложение объектом `AssemblyResolver` в ходе процесса, называемого *зондированием*. Этот объект вызывается платформой .NET Framework автоматически; программист не обязан вызывать его явно. Задача – определить имя сборки для EXE-программы и загрузить программу пользователя. В случае закрытой сборки объект `AssemblyResolver` ищет сборку только в каталоге загрузки приложения и в его подкаталогах (иными словами, в каталоге, из которого было вызвано приложение).

Три DLL-файла, созданные выше, должны находиться в том же каталоге, что и тестовая программа из примера 17.4 (или в одном из его подкаталогов).

Поставьте точку останова во второй строчке метода `Main()`, как показано на рис. 17.5.

```
Programming_CSharp.Test                    Main()
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        // main will not load the shared assembly
        static void Main()
        {
            Test t = new Test();
            t.UseCS();
            t.UseFraction();

        }
```

Рис. 17.5. *Точка останова в методе Main( )*

Выполните программу до точки останова и откройте окно `Modules`. Загружены только два модуля, что иллюстрирует рис. 17.6.

Если вы не разрабатывали *Test.cs* как часть решения Visual Studio .NET, поместите вызов `System.Diagnostics.Debugger.Launch()` непосредственно перед второй строкой метода `Main`. Это позволит выбрать, какой из отладчиков использовать. (Не забудьте, что *Test.cs* следует компилировать с ключами `/debug` и `/r:MySharedAssembly.dll`.)

Войдите внутрь первого метода, наблюдая за окном `Modules`. Как только будет вызван метод `UseCS()`, объект `AssemblyLoader` поймет, что требуется сборка из библиотеки *MySharedAssembly.Dll*. Библиотека DLL будет загружена, и из манифеста сборки объект `AssemblyLoader` узнает, что необходимо загрузить *Calc.dll*. Эта библиотека тоже загружается, как видно из рис. 17.7.

Когда отладчик зайдет внутрь метода `Fraction`, будет загружена третья DLL-библиотека. Достоинство многомодульных сборок в том и заключается, что модули загружаются только по мере необходимости.

Закрытые сборки

Сборки бывают двух видов: *закрытые* (*private*) и *совместно используемые* (*shared*). Закрытые сборки предназначены только для одного приложения; к совместно используемым могут обратиться сразу несколько приложений.

Modules

Name	Address	Path	Order	Version	Program
mscorlib.dll	6C660000-6C83C000	c:\winnt\microsoft.net\framework\v...	1	1.0.2728.2	[2280] Sha...
SharedAssem...	00400000-00408000	C:\ProgCSharp\SharedAssembly\bin...	2	1.0.535.29...	[2280] Sha...

Start Page | Test.cs

Programming_CSharp.Test | UseCS()

```
namespace Programming_CSharp
{
    using System;

    public class Test
    {
        // main will not load the shared assembly
        static void Main()
        {
            Test t = new Test();
            t.UseCS();
            t.UseFraction();

        }
```

Рис. 17.6. *Загружены только два модуля*

Modules

Name	Address	Path	Order	Version	Program
mscorlib.dll	6C660000-6C83C000	c:\winnt\microsoft.net\framework\v...	1	1.0.2728.2	[2280] Sha...
SharedAssem...	00400000-00408000	C:\ProgCSharp\SharedAssembly\bin...	2	1.0.535.29...	[2280] Sha...
mysharedass...	03010000-03018000	c:\progcsharp\sharedassembly\bin\d...	3	1.0.535.29...	[2280] Sha...
calc.dll	03030000-03036000	c:\progcsharp\sharedassembly\bin\d...	4		[2280] Sha...

Start Page | Test.cs

Programming_CSharp.Test | UseCS()

```
        }

        // calling this loads the myCalc assembly
        // and the mySharedAssembly assembly as well
        public void UseCS()
        {
            ProgCS.myCalc calc = new ProgCS.myCalc();
            Console.WriteLine("3+5 = {0}\n3*5 = {1}",
                calc.Add(3,5), calc.Mult(3,5));
        }
```

Рис. 17.7. *Модули загружаются по требованию*

До сих пор в этой главе строились только закрытые сборки. Когда компилируется приложение на языке C#, по умолчанию создается закрытая сборка. Файлы закрытой сборки хранятся в одном каталоге (или вложенных подкаталогах). Эти вложенные подкаталоги изолируются

от остальной системы, поскольку они доступны только одному приложению. Таким образом, для переноса этого приложения на другой компьютер достаточно скопировать каталог сборки со всеми подкаталогами.

Имя закрытой сборки может быть любым. Не страшно, если оно совпадает с именем сборки другого приложения, – эти имена локальны в рамках одного приложения.

Раньше при установке DLL-библиотек на компьютере в системном реестре Windows создавалась соответствующая запись. Избежать порчи реестра было трудно, и повторная установка программы на другом компьютере представляла собой нетривиальную задачу. Благодаря сборкам эти проблемы ушли в прошлое. Сборки сводят установку приложения к простому копированию. И все!

Совместно используемые сборки

Разработчик может создать сборку, доступную нескольким приложениям. Это требуется, например, в тех случаях, когда элемент управления или класс общего назначения применяется другими разработчиками. Если сборка предназначена для совместного использования, она должна удовлетворять определенным строгим требованиям.

Во-первых, у сборки должно быть *строгое имя* (*strong name*). Строгие имена являются глобально уникальными.

Никто другой, кроме разработчика сборки, не может создать точно такое же имя, поскольку сборка, созданная с одним закрытым ключом, гарантированно носит имя, отличное от имени любой сборки, созданной с другим закрытым ключом.

Во-вторых, сборка должна быть защищена от перезаписи новыми версиями. Иными словами, в ней должен присутствовать контроль версии.

И наконец, сборка должна быть помещена в *глобальный кэш сборок* (*Global Assembly Cache, GAC*). Это область файловой системы, отведенная средой CLR для хранения совместно используемых сборок.

Конец «кошмара DLL»

Сборки знаменуют собой конец «кошмара DLL». Читатель, конечно, помнит этот сценарий. На компьютере устанавливается приложение A, загружающее в каталог Windows некоторое количество DLL-библиотек. Несколько месяцев дела идут прекрасно, пока однажды на компьютере не появляется приложение B. Неожиданно приложение A перестает работать, хотя приложение B не имеет к нему никакого отношения. Что случилось? Со временем выясняется: приложение B заме-

нило DLL-библиотеку, необходимую приложению A, что и привело к полной беспомощности приложения A.

Когда были изобретены библиотеки динамической компоновки, пространство на диске считалось ценным ресурсом и многократное использование одной библиотеки DLL казалось удачным решением. Теоретически библиотеки DLL должны были обладать обратной совместимостью, чтобы автоматическая установка более свежей версии проходила безопасно и безболезненно. На память автору приходят слова его руководителя, Пэта Джонсона (Pat Johnson): «Теоретически теория и практика совпадают, но практически – никогда».

При установке на компьютере новой библиотеки DLL старое приложение, «тихо работавшее в своем углу», вдруг оказывалось подключенным к библиотеке, не соответствовавшей его ожиданиям. Результат, как правило, был печальным. Этот феномен вызывал у пользователей обоснованное недоверие к установке нового программного обеспечения и даже к обновлению уже существующего, что давало все основания считать операционную систему Windows нестабильной. С появлением сборок этот кошмар закончился.

Версии

Совместно используемые сборки уникально идентифицируются в .NET по своим именам и номерам версий. Область GAC позволяет старым и новым версиям одной сборки существовать одновременно и бесконфликтно. Таким образом, приложение может затребовать себе самую последнюю версию или, например, версию 2 последней компоновки, или ту версию, с которой оно было скомпоновано.

Сосуществование версий имеет отношение только к области GAC. Закрытые сборки не нуждаются в этой функциональной возможности и не обладают ею.

Номер версии для сборки выглядит как четыре числа, разделенные двоеточиями, например `1:0:2204:21`. Первые два числа – старший и младший номера версии. Третье число (`2204`) – номер компоновки, а четвертое (`21`) – номер выпуска.

Если у двух сборок отличаются старшие или младшие номера версий, то сборки считаются несовместимыми. Если у сборок различаются номера компоновок, то эти сборки могут быть, а могут и не быть совместимыми. Если же две сборки отличаются только разными номерами выпуска, то сборки считаются *безусловно* совместимыми.

Номера выпусков отражают процесс обнаружения и устранения ошибок. Если разработчик исправил ошибку и утверждает, что DLL обладает обратной совместимостью с существующей версией, он должен увеличить номер выпуска. Когда приложение загружает сборку, оно

указывает старший и младший номер версии, а объект `AssemblyResolver` ищет сборки с наибольшими номерами компоновки и выпуска.

Строгие имена

Чтобы загрузить совместно используемую сборку, необходимо соблюсти следующие правила:

- Точно указать, какая сборка требуется. Для этого необходимо знать глобальное уникальное имя сборки.
- Убедиться, что код сборки не подвергался постороннему вмешательству. Для этого требуется цифровая подпись, которой сборка была снабжена при компиляции и компоновке.
- Убедиться, что загружаемая сборка действительно создана заявленным производителем. Для этого необходимо провести идентификацию автора.

Всем этим требованиям удовлетворяют *строгие имена* (*strong name*). Строгие имена должны быть глобально уникальными, а для гарантии невмешательства в код и подлинности сборки применяется шифрование с открытым ключом. Строгое имя – это строка из шестнадцатеричных символов, понятная только компьютеру.

Чтобы создать строгое имя, необходимо создать для сборки пару «открытый/закрытый ключ». Имена и содержимое файлов сборки хеши-

Шифрование с открытым ключом

Строгие имена строятся по технологии шифрования с открытым ключом. Суть такого шифрования состоит в кодировании данных по сложной математической формуле, возвращающей два ключа. Данные, зашифрованные с помощью одного ключа, могут быть расшифрованы только с помощью другого. Данные, зашифрованные вторым ключом, можно расшифровать только первым.

Первый ключ предоставляется всем желающим и называется *открытым*. Второй закрытый ключ используется разработчиком и держится в секрете.

Отношение, установленное между ключами, позволяет зашифровать информацию открытым ключом так, что расшифровать ее можно будет только закрытым ключом. То есть после такой шифровки данные не сможет расшифровать никто, кроме владельца закрытого ключа, даже тот, кто их зашифровал.

Разработчик может зашифровать данные закрытым ключом так, что любой пользователь расшифрует их открытым ключом. Хотя подобное шифрование делает данные свободно доступными, существует гарантия, что их зашифровал именно этот разработчик. Это называется *цифровой подписью*.

руются, хеш шифруется закрытым ключом и помещается в манифест. Этот процесс называется *подписыванием сборки*. Открытый ключ встраивается в строгое имя сборки.

```
sn -k c:\myStrongName.snk
```

Когда приложение загружает сборку, среда CLR расшифровывает хеш файлов сборки с помощью открытого ключа. Расшифровка позволяет убедиться, что код не подвергся постороннему вмешательству, а также избежать конфликта имен.

Чтобы создать строгое имя, следует вызвать утилиту `sn`:

```
sn -k c:\myStrongName.snk
```

Флаг `-k` означает, что в указанный файл нужно записать новую пару ключей. Файл может носить любое допустимое имя. Помните, что строгое имя является строкой шестнадцатеричных символов, не предназначенной для человека.

Строгое имя связывается со сборкой при помощи атрибута:

```
using System.Runtime.CompilerServices
[assembly: AssemblyKeyFile("c:\myStrongName.key")]
```

Атрибуты подробно обсуждаются в главе 18. Просто поместите этот код в самое начало файла, чтобы связать со сборкой сгенерированное строгое имя.

Глобальный кэш сборок

После создания строгого имени и связывания его со сборкой остается лишь поместить сборку в глобальный кэш сборок, зарезервированный системный каталог. Это делается с помощью утилиты `gacutil`:

```
gacutil /i:MySharedAssembly.dll
```

Альтернативный способ заключается в перетаскивании сборки в окне Проводника (Windows Explorer). Чтобы увидеть область GAC, откройте Проводник и перейдите в каталог *%SystemRoot%\assembly*. Проводник превратится в утилиту просмотра GAC.

Компиляция совместно используемой сборки

Лучший способ понять, как работают совместно используемые сборки, – построить свою. Вернемся к ранее построенному многомодульному проекту (см. примеры с 17.1 по 17.4) и перейдем в каталог, где находятся файлы *calc.cs* и *fraction.cs*.

Поставим следующий эксперимент. Найдите каталог *bin* с тестовой программой и убедитесь в отсутствии локальных копий DLL-файлов сборки `MySharedAssembly`.

У сборки `MySharedAssembly` свойство `CopyLocal` должно иметь значение `false`.

Запустите программу. Она должна аварийно закончиться из-за вызова исключения, сообщающего о невозможности загрузить сборку:

```
Unhandled Exception: System.IO.FileNotFoundException: File or assembly name
MySharedAssembly, or one of its dependencies, was not found.
File name: "MySharedAssembly"
   at Programming_CSharp.Test.UseCS()
   at Programming_CSharp.Test.Main()
```

Теперь скопируйте DLL-файлы в один из подкаталогов каталога, содержащего тестовую программу. Запустите программу. Она будет работать.

Превратим `MySharedAssembly` в совместно используемую сборку. Во-первых, создадим для нее строгое имя, а во-вторых, поместим ее в GAC.

Этап 1: Создание строгого имени

Создайте пару ключей, открыв командное окно и введя:

```
sn -k keyFile.snk
```

Теперь в проекте для *MySharedAssembly.dll* откройте файл *AssemblyInfo.cs* и измените в нем строчку:

```
[assembly: AssemblyKeyFile("")]
```

следующим образом:

```
[assembly: AssemblyKeyFile(".\\keyFile.snk")]
```

Здесь для сборки указан файл ключей. Скомпонуйте программу с помощью make-файла, написанного ранее в этой главе. Откройте получившийся DLL-файл утилитой `ILDasm` и просмотрите манифест. Найдите открытый ключ, примерно такой, как на рис. 17.8.

```
.publickey = (00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00   // .$..............
              00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00   // .$..RSA1........
              11 13 95 3C 41 19 2B 41 28 29 E8 AF DE 8C A2 04   // ...<A.+A()......
              88 22 BD 4F A9 E1 F5 57 2C 2D E2 43 CF C3 68 4E   // ."..O...W,-.C..hN
              F7 C7 72 E8 55 94 85 11 EA 66 30 F6 D4 22 DB 0D   // ..r.U....f0.."..
              6E D6 A6 0D 6D 58 28 10 E9 75 D8 BF CC 82 2A EB   // n...mX(..u....*.
              04 19 D5 C1 86 B0 CF D5 CB E6 5C 58 43 08 0D E9   // ..........\XC...
              20 8F 9B DC 29 AC 46 A3 CD 7A 87 3C F8 92 7A 84   //  ...).F..z.<..z.
              E3 47 84 AD 56 73 3E 0D AD 1D C0 A2 FA 15 14 A3   // .G..Vs>.........
              88 17 01 AC F3 A3 7A F8 59 BC 3A 16 CB AB 34 C5 ) // ......z.Y.:...4.
```

Рис. 17.8. *Ключ разработчика в манифесте сборки MySharedAssembly.dll*

Добавление строгого имени эквивалентно постановке электронной подписи под сборкой (конкретное значение на компьютере читателя

будет другим). Теперь надо получить от DLL-библиотеки ее строгое имя. Для этого перейдите в каталог с этой библиотекой и введите команду:

```
sn -T MySharedAssembly.dll
```

Команда `sn` учитывает регистр символов, поэтому не вводите `sn -t`.

Результат будет примерно таким:

```
Public key token is 01fad8e0f0941a4d
```

Это значение является сокращенным вариантом открытого ключа, называемым *маркером открытого ключа*.

Удалите DLL-файлы из подкаталога программы и снова запустите ее. Она снова закончится аварийно. Хотя сборка получила строгое имя, она еще не зарегистрирована в глобальном кэше сборок.

Этап 2: Перенос совместно используемой сборки в GAC

Следующим шагом будет перенос библиотеки в GAC. Для этого откройте Проводник и перейдите в каталог *%SystemRoot%*. Если дважды щелкнуть по подкаталогу *assembly*, Проводник перейдет в режим просмотра GAC.

Библиотеку можно перенести в GAC, перетащив его мышью либо вызвав утилиту:

```
Gacutil -i mySharedAssembly
```

В любом случае необходимо убедиться, что сборка загружена в GAC и что значение, показанное в GAC для автора, совпадает со значением, возвращенным утилитой `sn`:

```
Public key token is 01fad8e0f0941a4d
```

Содержимое области GAC изображено на рис. 17.9.

Global Assembly Name	Type	Version	Culture	Public Key Token
Microsoft.Vsa		7.0.9188.0		b03f5f7f11d50a3a
Microsoft.Vsa		7.0.9174.0		b03f5f7f11d50a3a
Microsoft.Vsa.Vb.CodeDOMProcessor		7.0.0.0		b03f5f7f11d50a3a
msatinterop		1.0.0.0		826aaeb3f85826a0
mscorcfg		1.0.2411.0		b03f5f7f11d50a3a
mscorlib	PreJit	1.0.2411.0		b77a5c561934e089
MySharedAssembly		1.0.535.29377		a5929f0102e0c473

Рис. 17.9. *Область GAC*

Теперь совместно используемая сборка доступна любому пользователю. Обновите пользовательскую программу, перекомпоновав ее, и взгляните на манифест (рис. 17.10).

```
.assembly extern MySharedAssembly
{
  .publickeytoken = (A5 92 9F 01 02 E0 C4 73 )
  .ver 1:0:535:29377
}
```

Рис. 17.10. *Манифест*

Здесь сборка `MySharedAssembly` представлена как внешняя, а ее открытый ключ соответствует тому, что хранится в GAC. Очень хорошо; время, потраченное на эксперимент, не пропало даром.

Закройте `ILDasm`, скомпилируйте и запустите программу. Она должна работать корректно, несмотря на отсутствие файлов DLL-библиотек в каталоге. Причина тому – только что созданная совместно используемая сборка.

18

Атрибуты и отражение

По ходу изложения уже не раз говорилось, что приложение .NET содержит код, данные и метаданные. *Метаданные* – это информация о данных (то есть о типах, коде, сборке и т. д.), которая хранится вместе с программой. В этой главе будет показано, как создаются и используются некоторые метаданные.

Атрибуты (*attributes*) – это механизм добавления к программе таких метаданных, как инструкции компилятору и прочие сведения о данных, методах и классах. Атрибуты вставляются в метаданные и просматриваются с помощью специальных инструментов чтения метаданных, например `ILDasm`.

Отражение (*reflection*) – процесс, в ходе которого программа получает доступ к своим метаданным. Говорят, что программа использует отражение, когда она извлекает метаданные из сборки и использует их либо для информирования пользователя, либо для внесения изменений в собственное поведение.

Атрибуты

Атрибут – это объект, представляющий данные, которые разработчик хочет связать с элементом программы. Элемент, которому придается атрибут, называется его *целью*. Например, атрибут:

```
[NoIDispatch]
```

связывается с классом или интерфейсом и указывает на то, что при экспорте в модель COM целевой класс должен реализовать интерфейс `IUnknown`, а не интерфейс `IDispatch`. Программирование интерфейса COM подробно обсуждается в главе 22.

В главе 17 уже появлялся атрибут:

```
[assembly: AssemblyKeyFile("c:\myStrongName.key")]
```

Он помещает метаданные в сборку, чтобы указать строгое имя программы.

Стандартные атрибуты

Атрибуты бывают двух видов: *стандартные* (*intrinsic*) и *пользовательские* (*custom*). Стандартные атрибуты являются частью среды CLR; они интегрированы в .NET. *Пользовательские* атрибуты создаются программистом для собственных нужд.

Большинство программистов ограничивается использованием стандартных атрибутов, хотя пользовательские могут оказаться мощным инструментом, особенно в сочетании с механизмом отражения, описанным далее в этой главе.

Цели атрибутов

В среде CLR можно обнаружить множество атрибутов. Некоторые из них относятся к сборке, другие – к классу или интерфейсу, третьи, например [WebMethod], – к элементам класса. Все это – *цели атрибутов*. Допустимые цели атрибутов перечислены в табл. 18.1.

Таблица 18.1. Допустимые цели атрибутов

Имя	Использование
All	Применяется к любому из следующих элементов: сборка, класс, элемент класса, делегат, перечисление, событие, поле, интерфейс, метод, модуль, параметр, свойство, возвращаемое значение, структура
Assembly	Применяется к сборке в целом
Class	Применяется к объектам класса
Constructor	Применяется к конкретному конструктору
Delegate	Применяется к делегированному методу
Enum	Применяется к перечислению
Event	Применяется к событию
Field	Применяется к полю
Interface	Применяется к интерфейсу
Method	Применяется к методу
Module	Применяется к отдельному модулю
Parameter	Применяется к параметру метода
Property	Применяется к свойству (как к процедуре доступа get, так и к процедуре доступа set, если они реализованы)

Имя	Использование
ReturnValue	Применяется к возвращаемому значению
Struct	Применяется к структуре

Применение атрибутов

Атрибут заключают в квадратные скобки и ставят непосредственно перед целевым элементом. Атрибуты можно комбинировать, помещая их друг за другом:

```
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile(".\\keyFile.snk")]
```

Также можно перечислять их через запятую внутри квадратных скобок:

```
[assembly: AssemblyDelaySign(false),
 assembly: AssemblyKeyFile(".\\keyFile.snk")]
```

Атрибуты сборки должны находиться после всех операторов using, но до текста программы.

Многие стандартные атрибуты используются для взаимодействия с моделью COM; эта тема подробно обсуждается в главе 22. Один из атрибутов, [WebMethod], уже встречался читателю в главе 16, а другие, например [Serializable], будут применяться в ходе обсуждения сохранения объектов в потоке в главе 19.

Пространство имен System.Runtime предоставляет программисту целый ряд стандартных атрибутов, включая атрибуты для сборок (например keyname), для конфигурации (например, атрибут debug обозначает компоновку с отладочной информацией) и для контроля версий.

Стандартные атрибуты можно классифицировать по способам их использования. В основном они служат для взаимодействия с COM, для программного внесения изменений в файл IDL (Interface Definition Language, язык определения интерфейса), для взаимодействия с классами ATL Server и для выдачи указаний компилятору Visual C++.

По-видимому, программисты, пишущие на языке C#, чаще всего используют атрибут [Serializable] (конечно, если в их задачу входит взаимодействие с моделью COM). Как будет показано в главе 19, для сохранения класса на диске или в Интернете достаточно установить для этого класса атрибут [Serializable]:

```
[Serializable]
class MySerializableClass
```

Имя атрибута в квадратных скобках помещается непосредственно перед его целью, в данном случае перед объявлением класса.

Самое главное, что можно сказать о стандартных атрибутах, – программист знает, когда они нужны, поскольку их применение определяется самой решаемой задачей.

Пользовательские атрибуты

Программист волен определять собственные атрибуты и пользоваться ими на этапе исполнения программы по своему усмотрению. Пусть, например, руководство фирмы-производителя программного продукта желает отслеживать, как обнаруживаются и исправляются программные ошибки. Программист поддерживает базу данных исправленных ошибок, но хочет связать отчеты о них с конкретными фрагментами программы.

С этой целью можно вставлять в программу комментарий:

```
// ошибка 323 исправлена программистом Джесс Либерти 1/1/2005
```

Его легко прочитать в исходном тексте программы, но какая-либо связь с базой данных здесь отсутствует. Пользовательский атрибут – как раз то, что нужно в этой ситуации. Вместо комментария поставим:

```
[BugFixAttribute(323,"Jesse Liberty","1/1/2005"
Comment="Одной ошибкой меньше")]
```

Теперь можно написать программу, которая читает метаданные, ищет в них сообщения об исправлении ошибок и обновляет базу данных. Этот атрибут одновременно является и комментарием и источником информации для других программ.

Объявление атрибута

Как и большинство элементов языка C#, атрибуты представлены классами. Чтобы создать пользовательский атрибут, следует создать новый класс атрибута, производный от класса `System.Attribute`:

```
public class BugFixAttribute : System.Attribute
```

Необходимо сообщить компилятору, что является целью атрибута (то есть к каким элементам его можно применять). Эту информацию передает ему атрибут (а что же еще?):

```
[AttributeUsage(AttributeTargets.Class |
    AttributeTargets.Constructor |
    AttributeTargets.Field |
    AttributeTargets.Method |
    AttributeTargets.Property,
    AllowMultiple = true)]
```

Атрибут AttributeUsage является атрибутом, применяемым к другим атрибутам, и называется метаатрибутом. Он предоставляет, так сказать, метаметаданные, то есть данные о метаданных. Конструктор атрибута AttributeUsage имеет два аргумента. Первый аргумент является набором флагов-индикаторов цели; в рассматриваемом примере это класс и его конструктор, поля, методы и свойства. Второй аргумент представляет собой одиночный флаг, показывающий, может ли соответствующий элемент иметь более одного такого атрибута. Здесь флаг AllowMultiple имеет значение true, поэтому элементы класса могут иметь несколько атрибутов BugFixAttribute.

Имена атрибутов

Пользовательский атрибут, созданный в предыдущем примере, назван BugFixAttribute. Существует соглашение, по которому к имени атрибута добавляется суффикс Attribute. Компилятор осведомлен об этом соглашении и позволяет обращаться к атрибутам по укороченному имени, без суффикса. Так, можно написать:

```
[BugFix(123, "Jesse Liberty", "01/01/05", Comment=" Одной ошибкой меньше ")]
```

Вначале компилятор поищет атрибут с именем BugFix, затем, не найдя его, будет искать BugFixAttribute.

Конструкторы атрибута

Каждый атрибут должен иметь хотя бы один конструктор. Атрибуты имеют два вида аргументов: *позиционные* и *именованные*. В примере с атрибутом BugFix имя программиста и дата являются позиционными параметрами, а комментарий (Comment) – именованным. Позиционные параметры передаются атрибуту через конструктор, поэтому они должны быть указаны в порядке, определенном конструктором:

```
public BugFixAttribute(int bugID, string programmer,
string date)
{
    this.bugID = bugID;
    this.programmer = programmer;
    this.date = date;
}
```

Именованные параметры реализованы в виде свойств:

```
public string Comment
{
     get
     {
         return comment;
     }
```

```
        set
        {
            comment = value;
        }
    }
```

Общепринятой практикой является реализация позиционных параметров в виде свойств, доступных только для чтения:

```
    public int BugID
    {
        get
        {
            return bugID;
        }
    }
```

Применение атрибута

Определив атрибут, программист использует его, поставив непосредственно перед целью. Чтобы проверить, как действует атрибут BugFixAttribute из предыдущего примера, создадим простой класс по имени MyMath и определим в нем две функции. Атрибут BugFixAttribute будет использован в этом классе для ведения журнала исправленных ошибок:

```
    [BugFixAttribute(121,"Jesse Liberty","01/03/05")]
    [BugFixAttribute(107,"Jesse Liberty","01/04/05",
        Comment="Fixed off by one errors")]
    public class MyMath
```

Эти атрибуты будут храниться среди метаданных. В примере 18.1 приведен полный текст программы, демонстрирующий использование атрибута.

Пример 18.1. Работа с пользовательскими атрибутами

```
namespace Programming_CSharp
{
   using System;
   using System.Reflection;

   // создать пользовательский атрибут, применяемый к элементам класса
   [AttributeUsage(AttributeTargets.Class |
       AttributeTargets.Constructor |
       AttributeTargets.Field |
       AttributeTargets.Method |
       AttributeTargets.Property,
       AllowMultiple = true)]
   public class BugFixAttribute : System.Attribute
   {
      // конструктор атрибута для
      // позиционных параметров
```

```
public BugFixAttribute
   (int bugID,
   string programmer,
   string date)
{
   this.bugID = bugID;
   this.programmer = programmer;
   this.date = date;
}

// процедура доступа
public int BugID
{
   get
   {
      return bugID;
   }
}

// свойство для именованного параметра
public string Comment
{
   get
   {
      return comment;
   }
   set
   {
      comment = value;
   }
}

// процедура доступа
public string Date
{
   get
   {
      return date;
   }
}

// процедура доступа
public string Programmer
{
   get
   {
      return programmer;
   }
}

// закрытые переменные класса
private int     bugID;
private string  comment;
```

```
        private string  date;
        private string  programmer;
    }

    // ********* установка атрибутов класса ********

    [BugFixAttribute(121,"Jesse Liberty","01/03/05")]
    [BugFixAttribute(107,"Jesse Liberty","01/04/05",
        Comment="Fixed off by one errors")]
    public class MyMath
    {

        public double DoFunc1(double param1)
        {
            return param1 + DoFunc2(param1);
        }

        public double DoFunc2(double param1)
        {
            return param1 / 3;
        }

    }

    public class Tester
    {
        public static void Main()
        {
            MyMath mm = new MyMath();
            Console.WriteLine("Вызов DoFunc(7). Результат: {0}",
                mm.DoFunc1(7));
        }
    }
}
```

Вывод:

```
Вызов DoFunc(7). Результат: 9.3333333333333333
```

Как видно из примера, атрибуты не оказывают абсолютно никакого влияния на выводимую информацию. Более того, на данный момент читатель вынужден верить автору на слово, что атрибуты вообще существуют. Лишь просмотр метаданных с помощью ILDasm позволяет убедиться, что все атрибуты на месте (рис. 18.1). В следующем разделе показано, как извлекать метаданные и пользоваться ими в программе.

Отражение

Чтобы от атрибутов в метаданных была какая-нибудь польза, программисту нужен способ, позволяющий обращаться к ним, в идеале – на этапе выполнения. Классы из пространства имен Reflection, а также из пространств имен System.Type и System.TypedReference предоставляют поддержку процесса получения метаданных и взаимодействия с ними.

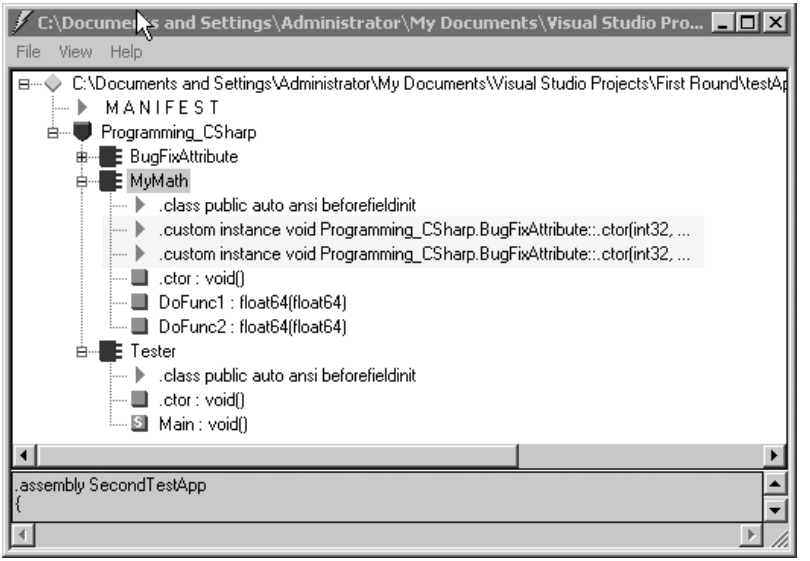

Рис. 18.1. *Метаданные в сборке*

Вообще говоря, отражение применяется для решения любой из следующих задач:

Просмотр метаданных

Просмотр выполняют инструментальные средства и утилиты, предназначенные для вывода метаданных на экран.

Получение информации по типам

Программист может исследовать типы в сборке и взаимодействовать с ними или создавать объекты этих типов, что бывает очень полезно при создании пользовательских сценариев. Например, можно разрешить пользователям взаимодействовать с программой при помощи языка создания сценариев, такого как JavaScript, или нестандартного языка, разработанного автором программы.

Позднее связывание с методами и свойствами

Позднее связывание позволяет программисту вызывать свойства и методы объектов, созданных динамически на основании исследования типов. Этот процесс иначе называется *динамическим вызовом*.

Создание типов на этапе выполнения (порождение отражения)

Последнее применение отражения состоит в создании новых типов на этапе исполнения и в последующем их использовании в программе. Обычно это делается в тех случаях, когда класс, созданный на этапе исполнения, работает быстрее, чем менее специфичный код, созданный во время компиляции. Пример будет приведен далее в этой главе.

Просмотр метаданных

В этом разделе поддержка отражения в C# будет использована для чтения метаданных класса MyMath.

Начнем с инициализации объекта типа MemberInfo. Этот объект предоставляется пространством имен System.Reflection и служит для чтения атрибутов элемента класса и обращения к метаданным:

```
System.Reflection.MemberInfo inf = typeof(MyMath);
```

Здесь вызывается метод typeof() класса MyMath, возвращающий объект типа Type, производный от класса MemberInfo.

Класс Type является базовым классом отражения. Он инкапсулирует представление типа объекта и является основным при обращении к метаданным. Type является производным от MemberInfo и инкапсулирует информацию об элементах класса (например, методах, свойствах, полях, событиях и т. д.).

На следующем шаге вызывается метод GetCustomAttributes() объекта MemberInfo, и тип искомого атрибута передается ему в качестве аргумента. Метод возвращает массив объектов, каждый из которых имеет тип BugFixAttribute:

```
object[] attributes;
attributes = inf.GetCustomAttributes(typeof(BugFixAttribute),false);
```

Теперь элементы массива можно просмотреть в цикле, выводя на экран свойства объекта BugFixAttribute. В примере 18.2 приведен фрагмент программы, которым следует заменить класс Tester из примера 18.1.

Пример 18.2. Применение отражения

```
public static void Main()
{
    MyMath mm = new MyMath();
    Console.WriteLine("Вызов DoFunc(7). Результат: {0}",
                      mm.DoFunc1(7));

    // получить информацию об элементе и воспользоваться ею
    // для чтения пользовательских атрибутов
    System.Reflection.MemberInfo inf = typeof(MyMath);
    object[] attributes;
    attributes =
       inf.GetCustomAttributes(typeof(BugFixAttribute), false);

    // просмотреть атрибуты в цикле,
    // читая их свойства
    foreach(Object attribute in attributes)
    {
        BugFixAttribute bfa = (BugFixAttribute) attribute;
```

```
            Console.WriteLine("\nBugID: {0}", bfa.BugID);
            Console.WriteLine("Programmer: {0}", bfa.Programmer);
            Console.WriteLine("Date: {0}", bfa.Date);
            Console.WriteLine("Comment: {0}", bfa.Comment);
        }
    }
```

Вывод:

```
Вызов DoFunc(7). Результат: 9.3333333333333333

BugID: 121
Programmer: Jesse Liberty
Date: 01/03/05
Comment:

BugID: 107
Programmer: Jesse Liberty
Date: 01/04/05
Comment: Fixed off by one errors
```

Поместите этот фрагмент программы в пример 18.1 и выполните его. Как и следовало ожидать, метаданные будут выведены на экран.

Получение информации о типах

Отражение можно использовать для изучения содержимого сборки. Программист может выяснить, какие типы связаны с модулем; какие методы, поля, свойства и события связаны с тем или иным типом, а также каковы сигнатуры методов этого типа; какие интерфейсы поддерживаются типом; наконец, каков базовый класс типа.

Для начала нужно динамически загрузить сборку статическим методом `Assembly.Load()`. Класс `Assembly` инкапсулирует сборку как раз для целей отражения, а метод `Load()` имеет следующую сигнатуру:

```
public static Assembly.Load(AssemblyName)
```

В следующем примере методу `Load()` передается библиотека ядра, *MsCorLib.dll*, которая содержит классы ядра платформы .NET Framework:

```
Assembly a = Assembly.Load("Mscorlib.dll");
```

Когда сборка будет загружена, можно вызывать метод `GetTypes()`, который возвратит массив объектов `Type`. Объект `Type` — самый главный объект отражения. Он представляет объявления типов (классов, интерфейсов, массивов, значений и перечислений):

```
Type[] types = a.GetTypes();
```

Сборка возвращает массив типов, который можно вывести на экран в цикле `foreach`, как показано в примере 18.3. Поскольку в этом примере используется класс `Type`, необходим оператор `using` для пространства имен `System.Reflection`.

Пример 18.3. Отражение сборки

```
namespace Programming_CSharp
{
    using System;
    using System.Reflection;

    public class Tester
    {
        public static void Main()
        {
            // содержимое сборки
            Assembly a = Assembly.Load("Mscorlib.dll");
            Type[] types = a.GetTypes();
            foreach(Type t in types)
            {
                    Console.WriteLine("Type is {0}", t);
            }
            Console.WriteLine("{0} types found", types.Length);
        }
    }
}
```

Вывод этой программы займет много страниц. Вот короткий отрывок:

```
Type is System.TypeCode
Type is System.Security.Util.StringExpressionSet
Type is System.Runtime.InteropServices.COMException
Type is System.Runtime.InteropServices.SEHException
Type is System.Reflection.TargetParameterCountException
Type is System.Text.UTF7Encoding
Type is System.Text.UTF7Encoding+Decoder
Type is System.Text.UTF7Encoding+Encoder
Type is System.ArgIterator
1426 types found
```

В этом примере массив заполняется информацией о типах, полученной из библиотеки ядра. На компьютере автора этот массив содержит 1426 элементов.

Отражение типа

Отражение можно выполнить и для одного типа из сборки `mscorlib`. Для этого нужно извлечь тип из сборки с помощью метода `GetType()`, как показано в примере 18.4.

Пример 18.4. Отражение типа

```
namespace Programming_CSharp
{
    using System;
    using System.Reflection;

    public class Tester
```

```
    {
        public static void Main()
        {
            // изучить отдельный объект
             Type theType =
                Type.GetType(
                   "System.Reflection.Assembly");
             Console.WriteLine(
                "\nSingle Type is {0}\n", theType);
        }
    }
}
```

Вывод:

```
Single Type is System.Reflection.Assembly
```

Получение всех членов данного типа

У типа `Assembly` можно запросить все его члены, вызвав метод `GetMembers()` класса `Type`. Этот метод возвратит все методы, свойства и поля, как показано в примере 18.5.

Пример 18.5. Отражение членов типа

```
namespace Programming_CSharp
{
   using System;
   using System.Reflection;

   public class Tester
   {
      public static void Main()
      {
         // изучить отдельный объект
         Type theType = Type.GetType("System.Reflection.Assembly");
         Console.WriteLine("\nSingle Type is {0}\n", theType);

         // получить все его члены
         MemberInfo[] mbrInfoArray = theType.GetMembers();
         foreach (MemberInfo mbrInfo in mbrInfoArray )
         {
            Console.WriteLine("{0} is a {1}", mbrInfo, mbrInfo.MemberType);
         }
      }
   }
}
```

Объем выводимой этой программой информации тоже очень велик. В нем перечисляются все поля, методы, конструкторы и свойства, что видно из следующего отрывка:

```
Boolean IsDefined(System.Type, Boolean) is a Method
System.Object[] GetCustomAttributes(Boolean) is a Method
System.Object[] GetCustomAttributes(System.Type, Boolean) is a Method
```

```
System.Security.Policy.Evidence get_Evidence() is a Method
System.String get_Location() is a Method
```

Поиск методов типа

Если читатель хочет сосредоточить внимание только на методах, исключив из рассмотрения поля, свойства и т. д., то он должен заменить вызов метода GetMembers() из предыдущего примера:

```
MemberInfo[] mbrInfoArray = theType.GetMembers(BindingFlags.LookupAll);
```

вызовом GetMethods():

```
mbrInfoArray = theType.GetMethods();
```

Теперь выводимая информация не содержит ничего, кроме методов:

Вывод (отрывок):

```
Boolean Equals(System.Object) is a Method
System.String ToString() is a Method
System.String CreateQualifiedName(System.String, System.String) is a Method
Boolean get_GlobalAssemblyCache() is a Method
```

Получение конкретных членов типа

Наконец, чтобы еще больше сузить область отражения, можно воспользоваться методом FindMembers(), возвращающим конкретные члены типа. Например, можно ограничить поиск лишь методами, имена которых начинаются с Get.

Метод FindMembers() имеет четыре параметра: MemberTypes, BindingFlags, MemberFilter и Object.

MemberTypes

Объект MemberTypes, указывающий тип искомого члена. Он принимает следующие значения: All, Constructor, Custom, Event, Field, Method, Nestedtype, Property и TypeInfo. Например, альтернативным способом поиска метода является MemberTypes.Method.

BindingFlags

Параметр перечислимого типа, управляющий способом поиска, проводимого отражением. Существует большое количество значений параметра BindingFlags, среди которых IgnoreCase, Instance, Public, Static и т. д.

MemberFilter

Делегат (см. главу 12), используемый для фильтрации списка элементов в массиве объектов MemberInfo. В рассматриваемом примере будет использовано поле Type.FilterName для фильтрации по имени.

Object

Строка, используемая фильтром. В данном случае это строка «Get*», определяющая методы, которые начинаются с букв Get.

Программа примера 18.6 демонстрирует использование этих методов.

Пример 18.6. Поиск конкретных элементов

```
namespace Programming_CSharp
{
   using System;
   using System.Reflection;

   public class Tester
   {
      public static void Main(  )
      {
         // изучить отдельный объект
         Type theType = Type.GetType(
            "System.Reflection.Assembly");

         // только элементы, являющиеся методами и начинающиеся с Get
         MemberInfo[] mbrInfoArray =
            theType.FindMembers(MemberTypes.Method,
               BindingFlags.Public |
               BindingFlags.Static |
               BindingFlags.NonPublic |
               BindingFlags.Instance |
               BindingFlags.DeclaredOnly,
               Type.FilterName, "Get*");
         foreach (MemberInfo mbrInfo in mbrInfoArray )
         {
            Console.WriteLine("{0} is a {1}", mbrInfo, mbrInfo.
                                 MemberType);
         }

         }
     }
}
```

Вывод (отрывок):

```
System.Type[] GetTypes(  ) is a Method
System.Type[] GetExportedTypes(  ) is a Method
System.Type GetType(System.String, Boolean) is a Method
System.Type GetType(System.String) is a Method
System.Reflection.AssemblyName GetName(Boolean) is a Method
System.Reflection.AssemblyName GetName(  ) is a Method
```

Позднее связывание

Найдя метод, его можно вызвать с помощью отражения. Пусть, например, требуется вызвать метод `Cos()` класса `System.Math`, возвращающий косинус угла.

Конечно, метод `Cos()` можно вызвать в программе и обычным способом, но отражение позволяет связаться с ним на этапе исполнения программы. Такая процедура называется *поздним связыванием*. Она позволяет выбирать связывае-

мый объект на этапе исполнения и обращаться к нему динамически. Позднее связывание оказывается полезным при создании пользовательского сценария, выполняемого пользователем, или при работе с объектами, недоступными на этапе компиляции. Например, применяя позднее связывание, программа может взаимодействовать с программой проверки орфографии или другим компонентом работающего текстового редактора, такого как Microsoft Word.

Чтобы вызвать метод Cos(), необходимо сначала получить информацию о типе для объекта класса System.Math:

```
Type theMathType = Type.GetType("System.Math");
```

Имея такую информацию, можно динамически загрузить экземпляр класса с помощью статического метода класса Activator. Поскольку Cos() — статический метод, нет необходимости создавать экземпляр System.Math (да это и невозможно, поскольку у System.Math нет открытого конструктора).

Класс Activator содержит четыре статических метода, применяемых для создания локальных или удаленных объектов или для получения ссылок на существующие объекты. Имена этих методов: CreateComInstanceFrom(), CreateInstanceFrom(), GetObject() и CreateInstance():

CreateComInstanceFrom()

Применяется для создания экземпляров COM-объектов.

CreateInstanceFrom()

Предназначен для создания ссылки на объект из конкретной сборки с указанным именем типа.

GetObject()

Применяется при маршалинге объектов. Маршалинг обсуждается в главе 19.

CreateInstance()

Используется для создания локального или удаленного экземпляра объекта.

Например:

```
Object theObj = Activator.CreateInstance(someType);
```

Вернемся к нашему примеру Cos(). Теперь у нас есть объект theMathType типа Type, который был создан при вызове метода GetType.

До вызова какого-либо метода объекта необходимо получить этот метод от объекта theMathType, имеющего тип Type. Для этого вызывается метод GetMethod(), которому передается сигнатура метода Cos().

Как помнит читатель, сигнатура — это имя метода (в данном случае Cos) и типы его аргументов. У метода Cos() только один аргумент,

имеющий тип double. Со своей стороны, метод Type.GetMethod() имеет два формальных параметра. Первый представляет имя требуемого метода, а второй – его аргументы. Имя передается в виде строки, аргументы – в виде массива объектов типа Type:

```
MethodInfo CosineInfo = theMathType.GetMethod("Cos",paramTypes);
```

Перед вызовом метода GetMethod() необходимо подготовить соответствующий массив:

```
Type[] paramTypes = new Type[1];
paramTypes[0]= Type.GetType("System.Double");
```

В этом фрагменте программы объявляется массив объектов типа Type, первый элемент которого (paramTypes[0]) заполняется объектом, представляющим тип double. Этот объект (имеющий тип Type) получен в результате вызова статического метода Type.GetType(), которому передана строка "System.Double".

Теперь у нас получен объект типа MethodInfo, и можно вызывать метод. Для этого надо передать ему объект, который будет им обрабатываться, и массив значений параметров. Поскольку Cos() – статический метод, можно передать theMathType. (Если бы Cos() был методом экземпляра, вместо theMathType можно было бы использовать theObj.)

```
Object[] parameters = new Object[1];
parameters[0] = 45* (Math.PI/180); // 45 градусов в радианах
Object returnVal = CosineInfo.Invoke(theMathType,parameters);
```

Обратите внимание, что было создано два массива. Первый, paramTypes, содержит типы аргументов, а второй, по имени parameters, хранит их фактические значения. Если бы метод принимал два аргумента, каждый из этих массивов имел бы два элемента. Если бы метод вообще не имел формальных параметров, массив все равно пришлось бы создать, указав для него нулевую длину!

```
Type[] paramTypes = new Type[0];
```

Как бы странно это ни выглядело, здесь все корректно.

Динамический вызов метода Cos() иллюстрируется примером 18.7.

Пример 18.7. Динамический вызов метода

```
namespace Programming_CSharp
{
   using System;
   using System.Reflection;

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

```
            Type theMathType = Type.GetType("System.Math");
            // Поскольку у System.Math нет открытого конструктора,
            // эта строчка вызовет исключение:
            //Object theObj = Activator.CreateInstance(theMathType);

            // Массив из одного элемента
            Type[] paramTypes = new Type[1];
            paramTypes[0]= Type.GetType("System.Double");

            // получить информацию о методе Cos()
            MethodInfo CosineInfo = theMathType.GetMethod("Cos",paramTypes);

            // заполнить массив фактическими значениями аргументов
            Object[] parameters = new Object[1];
            parameters[0] = 45 * (Math.PI/180); // 45 degrees in radians
            Object returnVal = CosineInfo.Invoke(theMathType,parameters);
            Console.WriteLine("The cosine of a 45 degree angle {0}", returnVal);
        }
    }
}
```

Такая громадная работа проделана лишь ради вызова одного метода. Однако мощь этой технологии заключается в том, что она позволяет применять отражение для поиска сборки на компьютере пользователя, выяснения списка доступных методов и вызова одного из них!

Динамическая генерация кода

До сих пор в этой главе отражение служило для трех целей: просмотра метаданных, получения информации о типе и динамического вызова методов. Эти приемы можно использовать при создании инструментальных средств (таких как среда разработки) или при обработке сценариев. Однако самое мощное применение отражения – это *динамическая генерация кода*, или *порождение отражения* (*reflection emit*).

Порождение отражения обеспечивает динамическое создание новых типов на этапе выполнения. Программист может определить сборку, которая будет динамически вызываться или сохраняться на диске, а также определять модули и новые типы с методами, вызываемыми динамически.

Динамический вызов и динамическая генерация кода являются, безусловно, сложными темами. Большинство программистов никогда не будут пользоваться динамической генерацией кода. Демонстрационная программа, приведенная в этом разделе, основывается на примере, представленном на конференции Microsoft Author's Summit, состоявшейся осенью 2000 года.

Чтобы понять всю силу порождения отражения, надо сначала разобрать чуть более сложный пример динамического вызова.

Для каждой задачи можно предложить общее решение, работающее сравнительно медленно, и специфическое решение, которое работает быстро. Чтобы не слишком усложнять пример, рассмотрим метод DoSum(), возвращающий сумму последовательности целых чисел от 1 до n, где n – число, вводимое пользователем.

Так, DoSum(3) возвращает 1+2+3, то есть 6, а DoSum(10) возвращает 55. На языке C# этот метод выглядит чрезвычайно просто:

```
public int DoSum1(int n)
{
    int result = 0;
    for(int i = 1;i <= n; i++)
    {
        result += i;
    }
    return result;
}
```

Он всего лишь выполняет цикл, складывая соответствующие числа. Если передать 3, метод сложит 1+2+3 и возвратит 6.

Для больших чисел и при неоднократном выполнении такое решение окажется медленным. Например, при n= 20 было бы гораздо эффективнее избавиться от цикла[1].

```
public int DoSum2()
{
    return 1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20;
}
```

Метод DoSum2() работает быстрее метода DoSum1(). Насколько быстрее? Чтобы выяснить это, в оба метода следует поместить таймеры. Воспользуемся объектом DateTime для отметки времени начала работы и объектом TimeSpan для вычисления интервала времени.

Чтобы поставить эксперимент, потребуется создать два метода DoSum(), первый с циклом, а второй – без. Каждый метод вызовем 1 000 000 раз. (Компьютеры работают очень быстро, и, чтобы разница была заметной, повторений должно быть много!) После этого сравним время выполнения. Вся тестовая программа содержится в примере 18.8.

Пример 18.8. Сравнение цикла и простого сложения

```
namespace Programming_CSharp
{
   using System;
   using System.Diagnostics;
```

[1] На самом деле, если говорить об оптимизации, то здесь лучше воспользоваться формулой для суммы арифметической прогрессии: (n+1)*n/2. – *Примеч. науч. ред.*

```
using System.Threading;

public class MyMath
{
   // сложить числа в цикле
   public int DoSum(int n)
   {
      int result = 0;
      for(int i = 1; i <= n; i++)
      {
         result += i;
      }
      return result;
   }

   // сложить вручную, "грубой силой"
   public int DoSum2()
   {
      return 1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20;
   }

}
public class TestDriver
{
   public static void Main()
   {

      const int val = 20;  // последнее число суммы

      // 1 000 000 итераций
      const int iterations = 1000000;

      // для хранения ответа
      int result = 0;

      MyMath m = new MyMath();

      // отметить время начала
      DateTime startTime = DateTime.Now;

      // поставить эксперимент
      for (int i = 0;i < iterations;i++)
      {
         result = m.DoSum(val);
      }

      // определить интервал времени
      TimeSpan elapsed =
         DateTime.Now - startTime;

      // вывести результаты
      Console.WriteLine(
         "Loop: Sum of ({0}) = {1}",
            val, result);
      Console.WriteLine(
         "The elapsed time in milliseconds is: " +
```

```
            elapsed.TotalMilliseconds.ToString());

         // отметить новое время начала
         startTime = DateTime.Now;

         // поставить эксперимент
         for (int i = 0;i < iterations;i++)
         {
            result = m.DoSum2();
         }

         // определить новый интервал времени
         elapsed = DateTime.Now - startTime;

         // вывести результаты
         Console.WriteLine(
            "Brute Force: Sum of ({0}) = {1}",
               val, result);
         Console.WriteLine(
            "The elapsed time in milliseconds is: " +
               elapsed.TotalMilliseconds);
      }
   }
}
```

Вывод:

```
Loop: Sum of (20) = 210
The elapsed time in milliseconds is: 187.5
Brute Force: Sum of (20) = 210
The elapsed time in milliseconds is: 31.25
```

Как видно из вывода, оба метода возвращают одно и то же число (миллион раз!), но метод «грубой силы» работает в шесть раз быстрее.[1]

Удастся ли обойтись без применения цикла, но все-таки предложить более общее решение? В традиционном программировании ответ будет отрицательным, однако отражение дает такую возможность. Вполне реальна программа, которая принимает от пользователя значения на этапе выполнения (в данном случае 20) и записывает на диск класс, реализующий метод «грубой силы». Затем этот метод вызывается динамически.

Такой результат можно получить по меньшей мере тремя способами, весьма элегантными. Лучше всех, конечно, порождение отражения, но рассмотрим сначала другие два, исключительно в учебных целях. Те, кто очень спешит, могут сразу перейти к разделу «Динамический вызов с помощью порождения отражения» далее в этой главе.

---

[1] Компиляторы обычно оптимизируют вычисления с константами еще на этапе компиляции. Поэтому «метод грубой силы», скорее всего, вообще ничего не вычисляет, а просто возвращает константное значение. – *Примеч. науч. ред.*

Динамический вызов с помощью метода InvokeMember()

Первый подход к описанной проблеме состоит в динамическом создании класса на этапе выполнения. Класс – назовем его BruteForceSums – будет содержать метод ComputeSum(), реализующий применение «грубой силы». Этот класс будет сохранен на диске и скомпилирован. Затем его метод будет вызван динамически с помощью метода InvokeMember() класса Type. Самым важным моментом при таком подходе является отсутствие файла *BruteForceSums.cs* до запуска программы. Он создается лишь при возникновении необходимости, после чего методу передаются аргументы.

Для реализации задуманного создадим класс с именем ReflectionTest. Его задача будет заключаться в создании класса BruteForceSums, записи его на диск и компиляции. У класса ReflectionTest только два метода: DoSum() и GenerateCode().

Метод ReflectionTest.DoSum() является открытым; он принимает значение и возвращает сумму. Так, если ему передать 10, результат будет 1+2+3+4+5+6+7+8+9+10. Для вычисления суммы он создает класс BruteForceSums и делегирует свою работу методу ComputeSum() этого класса.

Класс ReflectionTest содержит две закрытые переменные:

```
Type theType = null;
object theClass = null;
```

Первая представляеит собой объект типа Type и используется для загрузки класса с диска. Вторая переменная – это объект типа Object. Он служит для динамического вызова метода ComputeSum() класса BruteForceSums, который еще предстоит создать.

Демонстрационная программа создает экземпляр класса ReflectionTest и вызывает его метод DoSum(), передавая последнему значение, которое в данном примере будет равно 200.

Метод DoSum() сравнивает переменную theType со значением null. Если она имеет нулевое значение, то класс еще не создан. Метод DoSum() вызывает вспомогательный метод GenerateCode(), создающий исходный текст класса BruteForceSums, в частности его метода ComputeSums(). После этого GenerateCode() записывает созданный фрагмент программы в *.cs*-файл на диске и вызывает компилятор для превращения исходного текста в сборку. По окончании этого процесса метод DoSum() сможет вызывать необходимый ему метод, используя отражение.

Когда метод и класс созданы, сборка загружается с диска, и информация о типе класса присваивается переменной theType. Метод DoSum() на основании этой информации осуществляет динамический вызов метода, возвращающего искомую сумму.

Начнем с создания константы, имитирующей число, введенное пользователем:

```
const int val = 200;
```

При каждом выполнении программы будет вычисляться сумма чисел от 1 до 200.

Прежде чем создавать динамический класс, придется воссоздать класс MyMath():

```
MyMath m = new MyMath();
```

Включим в него метод DoSumLooping(), подобно тому как это делалось в предыдущем примере:

```
public int DoSumLooping (int initialVal)
{
    int result = 0;
    for(int i = 1;i <=initialVal;i++)
    {
        result += i;
    }
    return result;
}
```

Этот метод послужит эталоном, с которым будет сравниваться производительность метода «грубой силы».

Теперь все готово к созданию динамического класса и сравнению его производительности с версией, содержащей цикл. Вначале создадим объект типа ReflectionTest и вызовем его метод DoSum():

```
ReflectionTest t = new ReflectionTest();
result = t.DoSum(val);
```

Метод DoSum() проверяет, равна ли переменная theType (типа Type) значению null. Если это так, класс BruteForceSums еще не создан и тем более не скомпилирован. Это нужно сделать сейчас:

```
if (theType == null)
{
    GenerateCode(theValue);
}
```

Метод GenerateCode() принимает одно значение (в данном примере 200), сообщающее ему, сколько чисел требуется сложить.

Этот метод начинает свою работу с создания файла на диске. Файловый ввод/вывод подробно освещается в главе 21, а сейчас все пояснения будут весьма краткими. Вначале вызывается статический метод File.Open(), которому передаются имя файла и флаг, показывающий, что файл следует создать. Этот метод возвращает объект Stream:

```
string fileName = "BruteForceSums";
Stream s = File.Open(fileName + ".cs", FileMode.Create);
```

Имея такой объект, можно создать объект StreamWriter, позволяющий выводить информацию в файл:

```
StreamWriter wrtr = new StreamWriter(s);
```

Теперь для записи строк текста в файл можно вызвать метод WriteLine() объекта StreamWriter. Первой строчкой нового файла будет комментарий:

```
wrtr.WriteLine("// Dynamically created BruteForceSums class");
```

Этот оператор выводит текст:

```
// Dynamically created BruteForceSums class
```

в только что созданный файл *BruteForceSums.cs*. Далее выведем объявление класса:

```
string className = "BruteForceSums";
wrtr.WriteLine("class {0}", className);
wrtr.WriteLine("{");
```

Внутри класса определим метод ComputeSum():

```
wrtr.WriteLine("\tpublic double ComputeSum()");
wrtr.WriteLine("\t{");
wrtr.WriteLine("\t// Brute force sum method");
wrtr.WriteLine("\t// For value = {0}", theVal);
```

Настало время выводить оператор, вычисляющий сумму. По идее, это должна быть следующая строка кода:

```
return 0+1+2+3+4+5+6+7+8+9...
```

в которой вычисляется сумма от 0 до переданного значения (200).

```
wrtr.Write("\treturn 0");
for (int i = 1;i<=theVal;i++)
{
   wrtr.Write("+ {0}",i);
}
```

Разберем, как работает этот фрагмент программы. В файл выводится строка:

```
\treturn 0+ 1+ 2+ 3+...
```

Символы \t, стоящие в начале, при записи в файл транслируются в табуляторы.

По окончании работы цикла после оператора return ставится точка с запятой, а метод и класс завершаются фигурными скобками:

```
wrtr.WriteLine(";");
wrtr.WriteLine("\t}");
wrtr.WriteLine("}");
```

Закроем объект StreamWriter и поток, тем самым закрывая файл:

```
wrtr.Close();
s.Close();
```

После выполнения этой программы файл *BruteForceSums.cs* будет записан на диск. Его содержимое выглядит следующим образом:

```
// Dynamically created BruteForceSums class
class BruteForceSums
{
    public double ComputeSum()
    {
    // Brute force sum method

    // For value = 200

    return 0+ 1+ 2+ 3+ 4+ 5+ 6+ 7+ 8+ 9+ 10+
11+ 12+ 13+ 14+ 15+ 16+ 17+ 18+ 19+ 20+ 21+
22+ 23+ 24+ 25+ 26+ 27+ 28+ 29+ 30+ 31+ 32+
33+ 34+ 35+ 36+ 37+ 38+ 39+ 40+ 41+ 42+ 43+
44+ 45+ 46+ 47+ 48+ 49+ 50+ 51+ 52+ 53+ 54+
55+ 56+ 57+ 58+ 59+ 60+ 61+ 62+ 63+ 64+ 65+
66+ 67+ 68+ 69+ 70+ 71+ 72+ 73+ 74+ 75+ 76+
77+ 78+ 79+ 80+ 81+ 82+ 83+ 84+ 85+ 86+ 87+
88+ 89+ 90+ 91+ 92+ 93+ 94+ 95+ 96+ 97+ 98+
99+ 100+ 101+ 102+ 103+ 104+ 105+ 106+ 107+
108+ 109+ 110+ 111+ 112+ 113+ 114+ 115+ 116+
117+ 118+ 119+ 120+ 121+ 122+ 123+ 124+ 125+
126+ 127+ 128+ 129+ 130+ 131+ 132+ 133+ 134+
135+ 136+ 137+ 138+ 139+ 140+ 141+ 142+ 143+
144+ 145+ 146+ 147+ 148+ 149+ 150+ 151+ 152+
153+ 154+ 155+ 156+ 157+ 158+ 159+ 160+ 161+
162+ 163+ 164+ 165+ 166+ 167+ 168+ 169+ 170+
171+ 172+ 173+ 174+ 175+ 176+ 177+ 178+ 179+
180+ 181+ 182+ 183+ 184+ 185+ 186+ 187+ 188+
189+ 190+ 191+ 192+ 193+ 194+ 195+ 196+ 197+
198+ 199+ 200;
    }
}
```

Задача динамического создания класса с методом, вычисляющим сумму «грубой силой», решена.

Остается только скомпилировать и скомпоновать файл, а затем вызвать нужный метод. Для компиляции и компоновки необходимо запустить новый процесс (процессы обсуждаются в главе 20). Лучшим способом запуска процесса является использование структуры ProcessStartInfo, содержащей командную строку. Создайте экземпляр класса ProcessStartInfo, указав *cmd.exe* в качестве имени файла:

```
ProcessStartInfo psi = new ProcessStartInfo();
 psi.FileName = "cmd.exe";
```

Теперь необходимо передать содержимое командной строки. Свойство ProcessStartInfo.Arguments определяет аргументы командной строки, необходимые для запуска программы. Аргументом программы *cmd.exe* будет /c, который велит ей закончить работу после выполнения команды. За ним следует команда на компиляцию:

```
string compileString = "/c {0}csc /optimize+ ";
compileString += " /target:library ";
compileString += "{1}.cs > compile.out";
```

Строка compileString вызовет компилятор C# (*csc*) и сообщит ему о необходимости оптимизировать код (в конце концов, все это делается ради повышения производительности) и скомпоновать файл библиотеки динамической компоновки (*/target:library*). Результат работы компилятора помещается в файл с именем *compile.out*, который впоследствии будет проверен на наличие ошибок.

Имя исходного файла помещается в строку compileString с помощью статического метода Format() класса String, и сформированная строка присваивается свойству psi.Arguments. На место первого заполнителя ({0}) будет подставлена строка, соответствующая пути к компилятору (*%SystemRoot%\Microsoft.NET\ Framework\<version>*). Второй заполнитель ({1}) соответствует имени файла с исходным кодом:

```
string frameworkDir = RuntimeEnvironment.GetRuntimeDirectory();
psi.Arguments = String.Format(compileString, frameworkDir, fileName);
```

В результате всех этих действий свойство Arguments объекта psi класса ProcessStartInfo получает значения:

```
/c csc /optimize+ /target:library BruteForceSums.cs > compile.out
```

Прежде чем вызвать программу *cmd.exe*, необходимо установить свойство WindowStyle объекта psi в значение Minimized, чтобы во время выполнения команды ее окно не выводилось на экран пользователя:

```
psi.WindowStyle = ProcessWindowStyle.Minimized;
```

Теперь можно запускать процесс *cmd.exe*, причем следует подождать его окончания, прежде чем выполнять метод GenerateCode() дальше:

```
Process proc = Process.Start(psi);
proc.WaitForExit();
```

Когда процесс закончится, можно обратиться к сборке и прочитать из нее созданный класс. Наконец, у этого класса можно запросить тип и присвоить его переменной theType:

```
Assembly a = Assembly.LoadFrom(fileName + ".dll");
theClass = a.CreateInstance(className);
theType = a.GetType(className);
```

Созданный файл больше не нужен; удалим его:

```
File.Delete(fileName + ".cs");
```

Переменная theType заполнена, и все готово к вызову метода DoSum(), который, в свою очередь, вызовет метод ComputeSum() динамически. Объект Type реализует метод InvokeMember(), с помощью которого можно вызвать элемент класса, описываемый объектом Type. Метод InvokeMember() является перегруженным; здесь будет вызвана версия, принимающая пять аргументов:

```
public object InvokeMember(
   string name,
   BindingFlags invokeAttr,
   Binder binder,
   object target,
   object[] args
);
```

name

Имя вызываемого метода.

invokeAttr

Битовая маска типа BindingFlags, определяющая способ поиска объекта. В данном случае следует выполнить операцию «логическое ИЛИ» над флагами InvokeMethod и Default. Это стандартные флаги динамического вызова метода.

binder

Используется при преобразованиях типов. Передав значение null, программист соглашается на значение, установленное по умолчанию.

target

Объект, метод которого вызывается. В данном случае передается theClass, то есть класс, полученный из только что построенной сборки.

args

Массив аргументов, которые передаются вызываемому методу.

Полный вызов метода InvokeMember() выглядит так:

```
object[] arguments = new object[0];
object retVal =
    theType.InvokeMember("ComputeSum",
    BindingFlags.Default |
    BindingFlags.InvokeMethod,
    null,
    theClass,
    arguments);
return (double) retVal;
```

Результат, возвращаемый методом, присваивается локальной переменной retVal, которая возвращается вызывающей программе в виде значения типа double. Окончательный вариант программы приведен в примере 18.9.

Пример 18.9. Динамический вызов с помощью объекта Type и его метода InvokeMethod( )

```
namespace Programming_CSharp
{
   using System;
   using System.Diagnostics;
   using System.IO;
   using System.Reflection;
   using System.Runtime.InteropServices;

   // решение с циклом: используется в качестве эталона для сравнения
   public class MyMath
   {
      // сложить числа в цикле
      public int DoSumLooping(int initialVal)
      {
         int result = 0;
         for(int i = 1;i <=initialVal;i++)
         {
            result += i;
         }
         return result;
      }
   }

   // на этапе выполнения отвечает за создание
   // класса BruteForceSums, его компиляцию и
   // вызов метода DoSums
   public class ReflectionTest
   {
      // открытый метод, вызываемый тестовой программой
      public double DoSum(int theValue)
      {
         // если ссылка на динамически созданный класс
         // отсутствует, его нужно создать
         // create it
         if (theType == null)
         {
            GenerateCode(theValue);
         }

         // имея ссылку на динамически созданный класс,
         // можно вызывать метод
         object[] arguments = new object[0];
         object retVal =
            theType.InvokeMember("ComputeSum",
            BindingFlags.Default |
            BindingFlags.InvokeMethod,
            null,
            theClass,

            arguments);
```

```
        return (double) retVal;
    }

    // создать и скомпилировать программу
    private void GenerateCode(int theVal)
    {
        // открыть файл для записи
        string fileName = "BruteForceSums";
        Stream s =
            File.Open(fileName + ".cs", FileMode.Create);
        StreamWriter wrtr = new StreamWriter(s);
        wrtr.WriteLine("// Dynamically created BruteForceSums class");

        // создать класс
        string className = "BruteForceSums";
        wrtr.WriteLine("class {0}", className);
        wrtr.WriteLine("{");

        // создать метод
        wrtr.WriteLine("\tpublic double ComputeSum(  )");
        wrtr.WriteLine("\t{");
        wrtr.WriteLine("\t// Brute force sum method");
        wrtr.WriteLine("\t// For value = {0}", theVal);

        // записать формулу сложения
        wrtr.Write("\treturn 0");
        for (int i = 1;i<=theVal;i++)
        {
            wrtr.Write("+ {0}",i);
        }
        wrtr.WriteLine(";");    // завершить метод
        wrtr.WriteLine("\t}");  // закрывающая скобка метода
        wrtr.WriteLine("}");    // закрывающая скобка класса

        // закрыть объект и поток
        wrtr.Close(  );
        s.Close(  );

        // Скомпилировать и скомпоновать файл
        ProcessStartInfo psi = new ProcessStartInfo(  );
        psi.FileName = "cmd.exe";

        string compileString = "/c {0}csc /optimize+ ";
        compileString += "/target:library ";
        compileString += "{1}.cs > compile.out";

        string frameworkDir = RuntimeEnvironment.GetRuntimeDirectory(  );
        psi.Arguments = String.Format(compileString, frameworkDir,
            fileName);
        psi.WindowStyle = ProcessWindowStyle.Minimized;

        Process proc = Process.Start(psi);
        proc.WaitForExit(2000); // Подождать максимум 2 секунды
        // Открыть файл и получить указатель на информацию о методе
        Assembly a = Assembly.LoadFrom(fileName + ".dll");
```

```
            theClass = a.CreateInstance(className);
            theType = a.GetType(className);
            // File.Delete(fileName + ".cs");  // подчистка
         }
         Type theType = null;
         object theClass = null;
      }

      public class TestDriver
      {
         public static void Main(  )
         {
            const int val = 200;  // 1..200
            const int iterations = 100000;
            double result = 0;

            // запустить тест
            MyMath m = new MyMath(  );
            DateTime startTime = DateTime.Now;
            for (int i = 0;i < iterations;i++)
            {
               result = m.DoSumLooping(val);
            }
            TimeSpan elapsed =
               DateTime.Now - startTime;
            Console.WriteLine(
               "Sum of ({0}) = {1}",val, result);
            Console.WriteLine(
               "Looping. Elapsed milliseconds: " +
               elapsed.TotalMilliseconds +
               " for {0} iterations", iterations);

            // запустить альтернативный вариант
            ReflectionTest t = new ReflectionTest(  );

            startTime = DateTime.Now;
            for (int i = 0;i < iterations;i++)
            {
               result = t.DoSum(val);
            }

            elapsed = DateTime.Now - startTime;
            Console.WriteLine(
               "Sum of ({0}) = {1}",val, result);
            Console.WriteLine(
               "Brute Force. Elapsed milliseconds: " +
               elapsed.TotalMilliseconds  +
               " for {0} iterations", iterations);
         }
      }
   }
```

Вывод:

```
Sum of (200) = 20100
```

```
Looping. Elapsed milliseconds:
78.125 for 100000 iterations
Sum of (200) = 20100
Brute Force. Elapsed milliseconds:
3843.75 for 100000 iterations
```

Обратите внимание, что динамически вызванный метод работает *намного* медленнее эталонного. Неудивительно, ведь запись файла на диск, компиляция, чтение с диска и вызов метода требуют значительных расходов. Цель достигнута, но это пиррова победа.

Динамический вызов с помощью интерфейсов

Оказывается, динамический вызов работает очень медленно. Не станем отказываться от попыток создания и компиляции класса на этапе исполнения программы. Но вместо динамического вызова реализуем обычный вызов метода. Чтобы вызвать метод `ComputeSums()` непосредственно, воспользуемся интерфейсом.

С этой целью изменим метод `ReflectionTest.DoSum()` начиная отсюда:

```
public double DoSum(int theValue)
{
    if (theType == null)
    {
        GenerateCode(theValue);
    }
    object[] arguments = new object[0];
    object retVal =
        theType.InvokeMember("ComputeSum",
        BindingFlags.Default | BindingFlags.InvokeMethod,
        null,
        theFunction,
        arguments);
    return (double) retVal;
}
```

на следующий метод:

```
public double DoSum(int theValue)
{
    if (theComputer == null)
    {
        GenerateCode(theValue);
    }
    return (theComputer.ComputeSum());
}
```

В данном примере `theComputer` будет интерфейсом к объекту `BruteForceSums`. Это должен быть интерфейс, а не объект, поскольку при компиляции программы он еще не существует. Интерфейс `theComputer` создается динамически.

Удалите объявление переменных theType и theClass, заменив их на:

```
IComputer theComputer = null;
```

Здесь объявлено, что theComputer представляет собой интерфейс типа IComputer. В начале программы следует объявить сам интерфейс:

```
public interface IComputer
{
    double ComputeSum();
}
```

Создавая класс BruteForceSum, обязательно реализуйте в нем этот интерфейс:

```
wrtr.WriteLine(
"class {0} : Programming_CSharp.IComputer ", className);
```

Сохраните программу в файле проекта по имени Reflection и измените строку compileString в методе GenerateCode() следующим образом:

```
string compileString = "/c csc /optimize+ ";
compileString += "/r:\"Reflection.exe\" ";
compileString += "/target:library ";
compileString += "{0}.cs > compile.out";
```

Строка компиляции должна ссылаться на программу ReflectionTest (файл *Reflection.exe*), чтобы динамически вызванный компилятор знал, где искать объявление интерфейса IComputer.

После построения сборки объект не присваивается переменной theClass, поскольку ее больше нет. По аналогичной причине тип не присваивается переменной theType. Вместо этого присвоим объект интерфейса IComputer:

```
theComputer = (IComputer) a.CreateInstance(className);
```

Этот интерфейс служит для динамического вызова метода прямо из метода DoSum():

```
return (theComputer.ComputeSum());
```

Конечный вариант исходного текста программы приведен в примере 18.10.

Пример 18.10. Динамический вызов с помощью интерфейсов

```
namespace Programming_CSharp
{
   using System;
   using System.Diagnostics;
   using System.IO;
   using System.Reflection;
   using System.Runtime.InteropServices;
```

```
// решение с циклом используется в качестве эталона для сравнения
public class MyMath
{
   // сложить числа в цикле
   public int DoSumLooping(int initialVal)
   {
      int result = 0;
      for(int i = 1;i <=initialVal;i++)
      {
         result += i;
      }
      return result;
   }
}

public interface IComputer
{
   double ComputeSum(  );
}

// на этапе выполнения отвечает за создание
// класса BruteForceSums, его компиляцию
// и вызов метода DoSums
public class ReflectionTest
{
   // открытый метод, вызываемый программой
   public double DoSum(int theValue)
   {
      if (theComputer == null)
      {
         GenerateCode(theValue);
      }
      return (theComputer.ComputeSum(  ));
   }

   // создать и скомпилировать программу
   private void GenerateCode(int theVal)
   {
      // открыть файл для записи
      string fileName = "BruteForceSums";
      Stream s =
         File.Open(fileName + ".cs", FileMode.Create);
         StreamWriter wrtr = new StreamWriter(s);
         wrtr.WriteLine("// Dynamically created BruteForceSums class");

      // создать класс
      string className = "BruteForceSums";
      wrtr.WriteLine("class {0} : Programming_CSharp.IComputer ",
          className);
      wrtr.WriteLine("{");

      // создать метод
      wrtr.WriteLine("\tpublic double ComputeSum(  )");
```

```
        wrtr.WriteLine("\t{");
        wrtr.WriteLine("\t// Brute force sum method");
        wrtr.WriteLine("\t// For value = {0}", theVal);

        // записать формулу сложения
        wrtr.Write("\treturn 0");
        for (int i = 1;i<=theVal;i++)
        {
           wrtr.Write("+ {0}",i);
        }
        wrtr.WriteLine(";");      // завершить метод
        wrtr.WriteLine("\t}");    // закрывающая скобка метода
        wrtr.WriteLine("}");      // закрывающая скобка класса

        // закрыть объект и поток
        wrtr.Close(  );
        s.Close(  );

        // скомпилировать и скомпоновать файл
        ProcessStartInfo psi =
           new ProcessStartInfo(  );
        psi.FileName = "cmd.exe";

        string compileString = "/c {0}csc /optimize+ ";
        compileString += "/r:\"Reflection.exe\" ";
        compileString += "/target:library ";
        compileString += "{1}.cs > compile.out";

        string frameworkDir =
          RuntimeEnvironment.GetRuntimeDirectory(  );
        psi.Arguments =
           String.Format(compileString, frameworkDir, fileName);
        psi.WindowStyle = ProcessWindowStyle.Minimized;

        Process proc = Process.Start(psi);
        proc.WaitForExit(  );     // ждать 2 секунды

        // открыть файл и получить указатель
        // на информацию о методе
        Assembly a = Assembly.LoadFrom(fileName + ".dll");
        theComputer = (IComputer)  a.CreateInstance(className);
        File.Delete(fileName + ".cs");  // clean up
     }
     IComputer theComputer = null;
  }

  public class TestDriver
  {
     public static void Main(  )
     {
        const int val = 200;  // 1..200
        const int iterations = 1000000;
        double result = 0;

        // запустить тест
        MyMath m = new MyMath(  );
```

```
            DateTime startTime = DateTime.Now;
            for (int i = 0;i < iterations;i++)
            {
               result = m.DoSumLooping(val);
            }
            TimeSpan elapsed =
               DateTime.Now - startTime;
            Console.WriteLine(
               "Sum of ({0}) = {1}",val, result);
            Console.WriteLine(
               "Looping. Elapsed milliseconds: " +
               elapsed.TotalMilliseconds +
               " for {0} iterations", iterations);

            // запустить альтернативный рефлексивный вариант
            ReflectionTest t = new ReflectionTest(  );

            startTime = DateTime.Now;
            for (int i = 0;i < iterations;i++)
            {
               result = t.DoSum(val);
            }

            elapsed = DateTime.Now - startTime;
            Console.WriteLine("Sum of ({0}) = {1}",val, result);
            Console.WriteLine(
               "Brute Force. Elapsed milliseconds: " +
               elapsed.TotalMilliseconds  +
               " for {0} iterations", iterations);
         }
      }
   }
```

Вывод:

```
Sum of (200) = 20100
Looping. Elapsed milliseconds:
951.368 for 1000000 iterations
Sum of (200) = 20100
Brute Force. Elapsed milliseconds:
530.7632 for 1000000 iterations
```

Сейчас вывод выглядит вполне удовлетворительно. Динамически созданный метод, применяющий «грубую силу», работает примерно в два раза быстрее метода с циклом. Но еще лучшие результаты могут быть получены при использовании порождения отражения.

Динамический вызов с помощью порождения отражения

До сих пор сборка создавалась динамически путем записи исходного текста программы на диск с последующей ее компиляцией. Затем нужный метод динамически вызывался из этой сборки. Такая процедура

требовала огромных накладных расходов, но ради чего? Результатом записи файла на диск является исходный текст программы, который можно скомпилировать, а результатом его компиляции является код на языке IL, который можно выполнить на платформе .NET Framework.

Порождение отражения позволяет пропустить некоторые промежуточные шаги и порождать IL-код непосредственно. Код сборки создается прямо из программы на языке C#, после чего вызывается результат. Элегантнее не придумать!

Начнем почти так же, как начинали предыдущие примеры. Определим константы для количества складываемых чисел (200) и количества повторений (1 000 000). После этого создадим класс `myMath` для проведения эталонного теста.

По-прежнему определяем класс `ReflectionTest` и вызываем метод `DoSum()`, передавая ему количество складываемых чисел:

```
ReflectionTest t = new ReflectionTest();
result = t.DoSum(val);
```

Сам метод `DoSum()` практически не изменился:

```
public double DoSum(int theValue)
{
   if (theComputer == null)
   {
      GenerateCode(theValue);
   }

   // вызвать метод с помощью интерфейса
   return (theComputer.ComputeSum());
}
```

Как видим, снова используется интерфейс, но на этот раз файл на диск не записывается.

Текст метода `GenerateCode()` полностью переработан. Он больше не пишет файл на диск и не компилирует его. Вместо этого вызывается вспомогательный метод `EmitAssembly()`, который возвращает сборку. Затем создается объект этой сборки, который приводится к типу интерфейса:

```
public void GenerateCode(int theValue)
{
   Assembly theAssembly = EmitAssembly(theValue);
   theComputer = (IComputer) theAssembly.CreateInstance("BruteForceSums");
}
```

Как читатель уже догадался, самое интересное спрятано в методе `EmitAssembly()`:

```
private Assembly EmitAssembly(int theValue)
```

Значение, передаваемое методу, – это количество складываемых чисел. Чтобы ярче продемонстрировать порождение отражения, увеличим значение с 200 до 2000.

В первую очередь, в методе EmitAssembly() необходимо создать объект типа AssemblyName и назвать его "DoSumAssembly":

```
AssemblyName assemblyName = new AssemblyName();
assemblyName.Name = "DoSumAssembly";
```

Объект AssemblyName полностью описывает уникальную идентификацию сборки. Как было сказано в главе 17, сборка идентифицируется простым именем (DoSumAssembly), номером версии, парой криптографических ключей и поддерживаемой культурой.

Получив этот объект, можно создавать новый объект AssemblyBuilder, для чего следует вызвать метод DefineDynamicAssembly() текущего домена, получаемого за счет вызова статического метода GetDomain() объекта Thread. Домены подробно обсуждаются в главе 19.

Аргументами метода GetDomain() являются только что созданный объект AssemblyName и значение перечислимого типа AssemblyBuilderAccess (одно из трех: Run, RunandSave и Save). В данном случае передадим Run, чтобы пометить сборку как выполняемую, но не сохраняемую:

```
AssemblyBuilder newAssembly =
     Thread.GetDomain().DefineDynamicAssembly(assemblyName,
          AssemblyBuilderAccess.Run);
```

Создав объект AssemblyBuilder, можно создавать объект ModuleBuilder. Как нетрудно догадаться, его задача заключается в динамическом построении модуля. Модули обсуждались в главе 17. Вызываем метод DefineDynamicModule(), передавая ему имя метода, который требуется создать:

```
ModuleBuilder newModule = newAssembly.DefineDynamicModule("Sum");
```

Получив модуль, определим открытый класс и получим объект TypeBuilder. Класс TypeBuilder является корневым классом, используемым для управления динамическим созданием классов. Объект TypeBuilder позволяет определять классы и придавать им методы и поля:

```
TypeBuilder myType =
newModule.DefineType("BruteForceSums", TypeAttributes.Public);
```

Теперь все готово к тому, чтобы пометить новый класс как реализующий интерфейс IComputer:

```
myType.AddInterfaceImplementation(typeof(IComputer));
```

Можно создавать метод ComputeSum(), но сначала необходимо сформировать массив аргументов. Поскольку аргументы отсутствуют, создаем массив нулевой длины:

```
Type[] paramTypes = new Type[0];
```

Затем создаем объект типа Type для хранения типа, возвращаемого методом:

```
Type returnType = typeof(int);
```

Наконец, можно создавать требуемый метод. Метод DefineMethod() класса TypeBuilder не только создает метод, но также возвращает объект типа MethodBuilder, помогающий создать IL-код:

```
MethodBuilder simpleMethod =
myType.DefineMethod("ComputeSum",
                MethodAttributes.Public |
                   MethodAttributes.Virtual,
                returnType,
                paramTypes);
```

Здесь методу передаются следующие аргументы: имя создаваемого метода, атрибуты метода (public и virtual), тип возвращаемого значения (int) и массив нулевой длины (paramTypes).

Затем с помощью объекта MethodBuilder получаем объект ILGenerator:

```
ILGenerator generator = simpleMethod.GetILGenerator();
```

Получив этот драгоценный объект, можно порождать IL-коды. Это те самые коды, которые создал бы компилятор C#. (Вообще, лучший способ получить IL-код – это написать маленькую программу на языке C#, скомпилировать ее и изучить коды с помощью утилиты ILDasm!)

Вначале порождаем значение 0 и помещаем его в стек. Затем в цикле перебираем складываемые значения (от 1 до 200), причем каждое значение помещается в стек, после чего два верхних элемента стека складываются, а сумма помещается обратно в стек:

```
generator.Emit(OpCodes.Ldc_I4, 0);
for (int i = 1; i <= theValue;i++)
{
    generator.Emit(OpCodes.Ldc_I4, i);
    generator.Emit(OpCodes.Add);
}
```

Значение, оставшееся в стеке после всех сложений, и есть искомая сумма. Она возвращается вызвавшему методу:

```
generator.Emit(OpCodes.Ret);
```

Теперь пора создавать объект MethodInfo, описывающий требуемый код:

```
MethodInfo computeSumInfo = typeof(IComputer).GetMethod("ComputeSum");
```

Далее реализуем описанный метод. С этой целью вызовем метод DefineMethodOverride() объекта TypeBuilder, созданного ранее в программе. В качестве параметров передадим методу объекты MethodBuilder и MethodInfo:

```
myType.DefineMethodOverride(simpleMethod, computeSumInfo);
```

Дело почти сделано. Остается создать класс и возвратить сборку:

```
myType.CreateType();
return newAssembly;
```

Честно говоря, программа получилась непростой, но весьма элегантной и очень быстрой. Обычный цикл проделывает 1 000 000 итераций за 11,5 секунд, а порожденный код – за 0,4 секунды. На 3000% быстрее. Пример 18.11 содержит полный исходный текст этой замечательной программы.

Пример 18.11. Динамический вызов с помощью порождения отражения

```
namespace Programming_CSharp
{
   using System;
   using System.Diagnostics;
   using System.IO;
   using System.Reflection;
   using System.Reflection.Emit;
   using System.Threading;

   // решение с циклом используется в качестве эталона для сравнения
   public class MyMath
   {
      // сложить числа в цикле
      public int DoSumLooping(int initialVal)
      {
         int result = 0;
         for(int i = 1;i <=initialVal;i++)
         {
            result += i;
         }
         return result;
      }
   }

   // объявить интерфейс
   public interface IComputer
   {
      int ComputeSum();
   }

   public class ReflectionTest
   {
      // закрытый метод, порождающий сборку
      // с помощью IL-кодов
      private Assembly EmitAssembly(int theValue)
      {
         // сформировать имя сборки
         AssemblyName assemblyName =
            new AssemblyName();
         assemblyName.Name = "DoSumAssembly";
```

```
// создать одномодульную сборку
AssemblyBuilder newAssembly =
   Thread.GetDomain().DefineDynamicAssembly(
   assemblyName, AssemblyBuilderAccess.Run);
ModuleBuilder newModule =
   newAssembly.DefineDynamicModule("Sum");

//  объявить в сборке открытый класс
//  по имени BruteForceSums
TypeBuilder myType =
   newModule.DefineType(
   "BruteForceSums", TypeAttributes.Public);

// пометить, что класс реализует интерфейс IComputer
myType.AddInterfaceImplementation(
   typeof(IComputer));

// Определить метод вызываемого типа. Передать массив,
// определяющий типы аргументов, тип возвращаемого
// значения, а также имя и атрибуты метода.
Type[] paramTypes = new Type[0];
Type returnType = typeof(int);
MethodBuilder simpleMethod =
   myType.DefineMethod(
   "ComputeSum",
   MethodAttributes.Public |
   MethodAttributes.Virtual,
   returnType,
   paramTypes);

// Получить объект ILGenerator. Он используется
// для создания IL-кода.
ILGenerator generator =
   simpleMethod.GetILGenerator();

// Создать IL-код. Точно такой же код можно увидеть,
// если скомпилировать исходный текст,
// а затем вызвать ILDasm для просмотра результата.

// Положить ноль в стек. Для каждого i,
// не превышающего theValue, поместить i
// в стек, сложить два верхних элемента стека
// и поместить в стек сумму.
generator.Emit(OpCodes.Ldc_I4, 0);
for (int i = 1; i <= theValue;i++)
{
   generator.Emit(OpCodes.Ldc_I4, i);
   generator.Emit(OpCodes.Add);

}

// возвратить значение
generator.Emit(OpCodes.Ret);

// инкапсулировать информацию о методе и предоставить
```

```
        // доступ к метаданным метода
        MethodInfo computeSumInfo =
           typeof(IComputer).GetMethod("ComputeSum");

        // Определить реализацию метода.
        // Передать объект MethodBuilder, возвращенный
        // методом DefineMethod(), и объект methodInfo,
        // созданный в предыдущем операторе.
        myType.DefineMethodOverride(simpleMethod, computeSumInfo);

        // создать тип
        myType.CreateType();
        return newAssembly;
     }

     // сравнить интерфейс со значением null;
     // в случае равенства вызвать генератор
     public double DoSum(int theValue)
     {
        if (theComputer == null)
        {
           GenerateCode(theValue);
        }

        // вызвать метод через интерфейс
        return (theComputer.ComputeSum());
     }

     // породить сборку, создать экземпляр
     // и получить интерфейс
     public void GenerateCode(int theValue)
     {
        Assembly theAssembly = EmitAssembly(theValue);
        theComputer = (IComputer)
           theAssembly.CreateInstance("BruteForceSums");
     }

     // закрытые данные
     IComputer theComputer = null;

  }

  public class TestDriver
  {
     public static void Main()
     {
        const int val = 2000;  // Обратите внимание: 2000

        // 1 миллион итераций!
        const int iterations = 1000000;
        double result = 0;

        // запустить эталонный тест
        MyMath m = new MyMath();
        DateTime startTime = DateTime.Now;
```

```
            for (int i = 0;i < iterations;i++)
            {
               result = m.DoSumLooping(val);
            }
            TimeSpan elapsed =
               DateTime.Now - startTime;
            Console.WriteLine(
               "Sum of ({0}) = {1}",val, result);
            Console.WriteLine(
               "Looping. Elapsed milliseconds: " +
               elapsed.TotalMilliseconds +
               " for {0} iterations", iterations);

            // запустить альтернативный способ, использующий отражение
            ReflectionTest t = new ReflectionTest();

            startTime = DateTime.Now;
            for (int i = 0;i < iterations;i++)
            {
               result = t.DoSum(val);
            }

            elapsed = DateTime.Now - startTime;
            Console.WriteLine(
               "Sum of ({0}) = {1}",val, result);
            Console.WriteLine(
               "Brute Force. Elapsed milliseconds: " +
               elapsed.TotalMilliseconds  +
               " for {0} iterations", iterations);
         }
      }
   }
```

Вывод:

```
Sum of (2000) = 2001000
Looping. Elapsed milliseconds:
11468.75 for 1000000 iterations
Sum of (2000) = 2001000
Brute Force. Elapsed milliseconds:
406.25 for 1000000 iterations
```

Порождение отражения – это мощная технология программирования. Хотя современные компиляторы работают очень быстро, а современные компьютеры имеют огромную память и высокую производительность, приятно сознавать, что в случае необходимости можешь спуститься в самые недра «умной» машины.

19

Маршалинг и удаленные компоненты

Еще совсем недавно интегрированные программы исполнялись в одном процессе на одном компьютере. Однако эта эпоха закончилась или почти закончилась. Современные программы состоят из сложных компонентов, выполняемых в нескольких процессах, нередко на нескольких компьютерах одновременно. Всемирная паутина способствовала развитию распределенных приложений, более значительному, чем можно было представить себе еще несколько лет назад, причем тенденция к распределению ответственности становится все заметнее.

Другая тенденция ведет к централизации логики коммерческих программ на больших серверах. Хотя на первый взгляд эти тенденции противоречивы, в действительности они синергичны (то есть взаимно усиливают друг друга). Коммерческие объекты сосредотачиваются в отдельных центрах, а пользовательский интерфейс и даже часть вспомогательных программ распределяются по сети.

Как результат таких тенденций возникла необходимость в «общении» программных компонентов на расстоянии. Объекты, выполняемые на сервере, обрабатывающем запросы пользовательского интерфейса, должны быть в состоянии обмениваться информацией с коммерческими объектами на центральных серверах в штаб-квартирах больших корпораций.

Процедура переноса объекта через определенные границы называется *отдалением* (*remoting*). Границы могут существовать в программе на самых разных уровнях абстракции. Наиболее очевидная граница проходит между объектами, работающими на разных компьютерах.

Процедура подготовки объекта к переходу через границы называется *маршалингом* (*marshalling*). В пределах одного компьютера марша-

линг объектов может понадобиться для выхода за границы контекста, домена приложения или процесса.

Процесс – это, в сущности, работающее приложение. Если объект в текстовом редакторе должен взаимодействовать с объектом в электронной таблице, то им приходится общаться через границы процессов.

Процессы делятся на *домены приложений*, которые, в свою очередь, делятся на различные *контексты*. Домены приложений действуют как упрощенные процессы, а контексты создают границы, в которых объекты подчиняются единым правилам. Время от времени возникает необходимость передать объект через границу контекста, домена приложений, процесса и компьютера. (Процессы, домены приложений и контексты подробно обсуждаются далее в этой главе.)

При отдалении объекта (передаче его через границы) создается впечатление, что он посылается по линии связи от одного компьютера другому, как капитан Кирк из «Звездного пути» телепортировался на поверхность планеты из космического корабля «Энтерпрайз», находившегося на орбите.

В фильме Кирк действительно оказывается на планете, однако если бы сюжет развивался в среде .NET, все это было бы чистой иллюзией. Те, кто находился на планете, думали бы, что разговаривают с самим Кирком, но это был бы не он, а его заместитель (proxy), то есть имитация, в задачу которой входит выслушать собеседника и передать его слова Кирку, который остался на космическом корабле. Между обитателями планеты и Кирком находился бы также целый ряд «получателей».

Получатель (*sinks*) – это объект, в задачу которого входит проведение определенной политики. Например, если Кирк скажет что-нибудь, способное изменить ход развития цивилизации на планете, получатель первичных команд (prime-directive sink) просто запретит передачу.

Когда Кирк отвечает жителям планеты, его слова проходят через последовательность получателей, пока не достигнут заместителя, который и «озвучит» сообщение. Те, кто разговаривает с ним, и не подозревают, что на самом деле он стоит на капитанском мостике и кричит Скотти, что ему не хватает мощности.

Собственно передача сообщений выполняется через *канал*. Канал «знает», как пересылать сообщение с космического корабля на планету. Канал работает с *форматизатором* (*formatter*), гарантирующим, что сообщение будет иметь соответствующий формат. Предположим, на планете все говорят на вулканском языке, а капитан его не понимает. Форматизатор преобразует сообщение в Стандарт Федерации, а ответ Кирка переведет на вулканский язык. Собеседники и не замечают, как форматизатор тихо делает свою работу.

В этой главе демонстрируется, как объекты составляются для передачи через различные границы и как заместители и заглушки создают иллюзию присутствия объекта на компьютере, словно он пришел по

проводам из соседнего офиса или даже с другого конца земного шара. Кроме того, в этой главе раскрывается роль форматизаторов, каналов и получателей и демонстрируется применение этих понятий в программировании.

Домены приложений

Процесс – это, в сущности, работающее приложение. Каждое приложение .NET выполняется в своем процессе. Если на компьютере открыты Word, Excel и Visual Studio .NET, значит, на нем работают как минимум три процесса. Если открыть еще один экземпляр редактора Word, будет запущен четвертый процесс.

Каждый процесс состоит из одного или нескольких *доменов приложений*. Домен приложения работает аналогично процессу, но занимает меньше ресурсов.

Домены приложений запускаются и останавливаются независимо друг от друга. Они безопасны и обладают «облегченной» и многосторонней функциональностью. Домены делают приложение устойчивым к сбоям. Если объект в одном из доменов приложения приведет к краху, то прервется исполнение только этого домена, а не всей программы. Нетрудно догадаться, что веб-серверам домены приложений нужны для выполнения пользовательских программ. Если программа столкнется с проблемами, на работе всего сервера это не отразится.

Домен приложения инкапсулируется экземпляром класса `AppDomain`, обладающего рядом свойств и методов. Наиболее важные из них перечислены в табл. 19.1.

Таблица 19.1. Свойства и методы класса AppDomain

Метод или свойство	Описание
`CurrentDomain`	Открытое статическое свойство, возвращающее текущий домен приложения для текущего вычислительного потока
`CreateDomain()`	Перегруженный открытый статический метод, создающий новый домен приложения
`GetCurrentThreadID()`	Открытый статический метод, возвращающий идентификатор текущего вычислительного потока
`Unload()`	Открытый статический метод, удаляющий указанный домен приложения
`FriendlyName`	Открытое свойство, возвращающее дружественное имя домена приложения
`DefineDynamicAssembly()`	Перегруженный открытый метод, определяющий динамическую сборку в текущем домене приложения
`ExecuteAssembly()`	Открытый метод, выполняющий указанную сборку

Таблица 19.1 (продолжение)

Метод или свойство	Описание
GetData()	Открытый метод, возвращающий значение из текущего домена приложения по заданному ключу
Load()	Открытый метод, загружающий сборку в текущий домен приложения
SetAppDomainPolicy()	Открытый метод, проводящий политику безопасности в текущем домене приложения
SetData()	Открытый метод, записывающий данные в указанное свойство домена приложения

Домены приложений поддерживают целый ряд событий, в том числе AssemblyLoad, AssemblyResolve, ProcessExit и ResourceResolve, которые возникают по мере того, как сборки обнаруживаются, загружаются, выполняются и выгружаются.

Каждый процесс имеет начальный домен приложения и может обладать дополнительными, которые создаются по воле программиста. Любой домен приложения существует только в одном процессе. До сих пор в этой книге все программы выполнялись в одном домене приложения, а именно в том, который устанавливается по умолчанию.

Домен приложения, устанавливаемый по умолчанию, есть у каждого процесса. Для многих, может быть, для всех программ, которые напишет читатель, будет достаточно домена приложения, устанавливаемого по умолчанию.

Впрочем, бывают ситуации, в которых одного домена недостаточно. Например, удобно иметь второй домен для исполнения библиотеки, созданной другим программистом. Если к этой библиотеке нет полного доверия, логично изолировать ее в отдельном домене. Если какой-то метод из библиотеки окажется ненадежным, то «рухнет» только изолированный домен. Если бы автор этой книги разрабатывал IIS (Internet Information Server, веб-сервер фирмы Microsoft), он создавал бы новый домен приложения для каждого добавляемого модуля и каждого виртуального каталога. Это гарантировало бы его устойчивость в случае краха того или иного веб-приложения.

Возможны также случаи, когда второй библиотеке требуется какая-то другая среда безопасности. Создание второго домена позволит двум средам безопасности сосуществовать в одной программе. Каждый домен приложения имеет свою защиту, а границами защиты являются границы домена.

Домены приложений следует отличать от вычислительных потоков. Вычислительный поток существует в конкретный момент времени в одном домене приложения, причем поток всегда может узнать (и сообщить), в каком домене приложения он сейчас выполняется. Домены приложений служат для изоляции программ, а в рамках одного домена

в любой момент времени может работать несколько потоков (см. главу 20).

Чтобы понять, как работают домены приложений, разберем пример. Предположим, программа должна создать экземпляр класса Shape, но не в текущем, а в другом домене приложения.

Нет никаких причин помещать класс Shape в отдельный домен приложения, кроме желания автора продемонстрировать эту технологию программирования. Однако бывают случаи, когда сложные объекты нуждаются во втором домене приложения, который предоставил бы им специфическую среду безопасности. Кроме того, когда программист создает классы, которые могут быть вовлечены в рискованные ситуации, он, скорее всего, запустит их в отдельном домене приложения, чтобы защитить более надежный код.

В обычной ситуации класс Shape загружается из отдельной сборки, но ради простоты поместим определение класса Shape в тот же исходный файл, что и остальной код примера (см. главу 17). Кроме того, при разработке реального приложения методы класса Shape выполнялись бы в отдельном потоке. Сейчас, опять-таки для упрощения примера, проигнорируем эту возможность. (Потоки подробно обсуждаются в главе 20.) Если отбросить эти непринципиальные моменты, можно написать простой пример, позволяющий сосредоточиться на создании и использовании доменов приложений, а также на передаче объектов через их границы.

Создание и использование доменов приложений

Новый домен создается статическим методом CreateDomain() класса AppDomain:

```
AppDomain ad2 = AppDomain.CreateDomain("Shape Domain");
```

Эта строка создает новый домен приложения с *дружественным именем* Shape Domain. Дружественное имя необходимо для удобства программиста. По этому имени можно обратиться к домену из программы, даже ничего не зная о внутреннем представлении домена. Чтобы узнать дружественное имя текущего домена, следует прочитать свойство System.AppDomain.CurrentDomain.FriendlyName.

Получив объект класса AppDomain, можно создавать объекты классов, интерфейсов и т. д., вызывая метод CreateInstance(). Вот его сигнатура:

```
public ObjectHandle CreateInstance(
   string assemblyName,
   string typeName,
   bool ignoreCase,
   BindingFlags bindingAttr,
   Binder binder,
```

```
    object[] args,
    CultureInfo culture,
    object[] activationAttributes,
    Evidence securityAttributes
);
```

А вот пример его вызова:

```
ObjectHandle oh = ad2.CreateInstance(
"ProgCSharp",                                   // имя сборки
"ProgCSharp.Shape",                            // тип имени в пространстве имен
false,                                          // игнорировать регистр
System.Reflection.BindingFlags.CreateInstance, // флаг
null,                                           // связывание сборки
new object[] {3, 5},                            // аргументы
null,                                           // культура
null,                                           // атрибуты вызова
null );                                         // атрибуты безопасности
```

Первый параметр, `ProgCSharp`, является именем сборки, а второй, `ProgCSharp.Shape`, именем класса. Имя класса должно быть полностью квалифицировано пространствами имен.

Пятый параметр (класса `Binder`) является объектом, выполняющим динамическое связывание сборки на этапе выполнения. В число его задач входит получение информации о создаваемом объекте, собственно создание объекта и установка ссылки на этот объект (то есть связывание с ним). В подавляющем большинстве случаев, включая данный пример, программиста устраивает значение, установленное по умолчанию, о чем он сообщает, передавая методу значение `null`.

Конечно, всегда можно написать собственный класс для динамического связывания, который, например, будет искать в базе данных идентификационный номер программиста и выполнять связывание с объектами на основании записанных там привилегий.

Обычно термин «связывание» означает закрепление за объектом некоторого имени. «Динамическое связывание» — это закрепление имени на этапе исполнения программы, а не во время компиляции. В рассматриваемом примере объект `Shape` связывается с переменной объекта на этапе исполнения программы при помощи метода `CreateInstance()` текущего домена приложения.

Флаги связывания помогают связывающему классу точнее настроить свое поведение. В рассматриваемом примере используется значение `CreateInstance` перечислимого типа `BindingFlags`. По умолчанию связывающий класс ищет только открытые классы, но при наличии соответствующих прав доступа программист может установить флаги, заставляющие связывающий класс искать закрытые классы.

Если сборка связывается на этапе выполнения, она не указывается на этапе компиляции. Когда программа выполняется, она определяет, какая сборка нужна, и связывает с ней соответствующую переменную.

Конструктор, вызываемый в этом примере, принимает в качестве аргументов два целых значения, которые должны быть помещены в массив типа `object` (`new object[] {3, 5}`). В качестве аргумента «региональные настройки» можно указать `null`, что означает региональные настройки, установленные по умолчанию (`en`). Атрибуты вызова и безопасности не уточняются.

Метод `CreateInstance()` возвращает *дескриптор объекта*. Это специальный тип, называемый `ObjectHandle`, используемый для передачи объекта (в обернутом виде) из одного домена приложения в другой без загрузки метаданных передаваемого объекта в каждом объекте, через который он передается. Чтобы получить собственно объект, созданный методом, следует вызвать метод `UnWrap()` для дескриптора объекта, а результат привести к соответствующему типу, в данном случае `Shape`.

Метод `CreateInstance()` позволяет программисту создавать объект в новом домене приложения. Если бы объект создавался с модификатором `new`, он появился бы в текущем домене приложения.

Маршалинг через границы домена приложения

Итак, объект `Shape` создан в домене `Shape Domain`, однако обращаться к нему можно через объект `Shape` в исходном домене. Чтобы обратиться к объекту в другом домене, его надо *маршалировать* через границы домена.

Маршалинг (*marshaling*) – это процедура подготовки объекта к переходу через границы домена, аналогичная подготовке капитана Кирка к телепортации на поверхность чужой планеты. Маршалинг выполняется двумя способами: *по значению* или *по ссылке*. Маршалинг по значению передает через границу копию объекта. Проведем следующую аналогию. Пусть читатель представит, что он позвонил приятелю и попросил одолжить ему калькулятор, а тот идет в магазин, покупает точно такой же калькулятор, какой лежит у него дома, и просит доставить его по адресу читателя. На этой копии можно производить вычисления, но нажатие на клавиши никак не отражается на оригинальном калькуляторе, который остается у приятеля.

Маршалинг по ссылке, в определенном смысле аналогичен передаче своего калькулятора, но здесь есть некоторые тонкости. На самом деле хозяин калькулятора передает не сам приборчик, а посредника. Если нажать кнопку на заместителе, он пошлет сигнал исходному калькулятору, на дисплее которого появится цифра. У того, кто работает с заместителем калькулятора, создается впечатление, что он работает с исходным калькулятором.

Маршалинг и заместители

Капитан Кирк и калькуляторы хороши ровно настолько, насколько хороши могут быть аналогии. Что же фактически происходит при маршалинге по ссылке? Среда CLR предоставляет вызывающему объекту механизм, называемый *прозрачным заместителем*.

Задача прозрачного заместителя состоит в изъятии из стека информации, связанной с вызовом метода (то есть возвращаемого значения, аргументов и т. д.) и упаковки ее в объект, реализующий интерфейс `IMessage`. Этот объект передается затем объекту `RealProxy`.

`RealProxy` является абстрактным базовым классом, потомками которого являются все заместители. Программист может реализовать собственный заместитель или любой из других объектов, участвующих в этой процедуре, но не прозрачный заместитель. По умолчанию объект `RealProxy` передаст `IMessage` последовательности объектов получателей.

Программист может использовать любое число получателей в зависимости от проводимой политики, однако последний получатель в цепочке должен поместить объект `IMessage` в объект `Channel` (канал). Различают каналы пользователя и сервера, а их работа состоит в передаче сообщения через границу. Каналы должны понимать протокол передачи. Формат сообщения, передаваемого через границу, контролируется *форматизатором*. Платформа .NET Framework предоставляет два форматизатора: SOAP, устанавливаемый по умолчанию для каналов HTTP, и Binary, который используется по умолчанию для каналов TCP/IP. Программист волен создать собственный форматизатор и, если он настоящий мазохист, собственный канал.

Когда сообщение пересечет границу, оно будет принято серверным каналом и форматизатором, которые преобразуют объект `IMessage` и передадут его одному или нескольким получателям на сервере. Последним получателем в цепочке является объект `StackBuilder`, ответственный за принятие объекта `IMessage` и перевод его во фрейм стека так, что на сервере сообщение предстанет вызовом функции.

Задание метода маршалинга

Для иллюстрации различия между маршалингом по значению и маршалингом по ссылке в следующем примере выполняется маршалинг по ссылке для объекта `Shape`. Однако в нем определяется переменная типа `Point`, для которой выполняется маршалинг по значению.

Обратите внимание, что каждый раз, когда создается объект, который может быть передан через границу, программист должен выбрать способ маршалинга. В нормальной ситуации объекты вообще не могут маршалироваться. Чтобы показать, что какой-либо объект может маршалироваться (по ссылке либо по значению), программист должен предпринять определенные действия.

Простейший способ превратить объект в маршалируемый по значению сводится к указанию атрибута Serializable:

```
[Serializable]
public class Point
```

Когда объект сериализуется, его внутреннее состояние выводится в поток данных либо для маршалинга, либо для сохранения. Подробности процедуры сериализации приведены в главе 21.

Простейший способ указать, что объект маршалируется по ссылке, заключается в выборе для него базового класса MarshalByRefObject:

```
public class Shape : MarshalByRefObject
```

У класса Shape будет еще одна переменная – upperLeft. Это объект класса Point, в котором хранятся координаты левого верхнего угла фигуры, представляемой объектом Shape.

Переменная Point инициализируется в конструкторе Shape:

```
public Shape(int upperLeftX, int upperLeftY)
{
    Console.WriteLine( "[{0}] Event{1}",
       System.AppDomain.CurrentDomain.FriendlyName,
       "Shape constructor");
    upperLeft = new Point(upperLeftX, upperLeftY);
}
```

Добавим в класс Shape метод, возвращающий положение объекта:

```
public void ShowUpperLeft()
{
    Console.WriteLine( "[{0}] Upper left: {1},{2}",
        System.AppDomain.CurrentDomain.FriendlyName,
        upperLeft.X, upperLeft.Y);
}
```

Кроме того, напишем метод, возвращающий переменную upperLeft:

```
public Point GetUpperLeft()
{
    return upperLeft;
}
```

Класс Point тоже очень прост. Он содержит конструктор, инициализирующий две переменные, и методы, возвращающие их значения.

Создав объект Shape, запросим его координаты:

```
s1.ShowUpperLeft();     // запросить у объекта его положение
```

Теперь попросим его возвратить координаты upperLeft в виде объекта типа Point, который можно будет изменить:

```
Point localPoint = s1.GetUpperLeft();
```

```
localPoint.X = 500;
localPoint.Y = 600;
```

Теперь пусть объект Point выведет на экран свои координаты, а объект Shape – свои. Отразятся ли изменения в локальном объекте Point на объекте Shape? Все зависит от способа маршалинга объекта Point. Если он был передан по значению, локальный объект localPoint будет копией, изменение которой не затронет объект Shape. С другой стороны, если объект Point передавался по ссылке, то программа имеет дело с заместителем объекта upperLeft. Когда он изменяется, объект Shape изменяется *вместе с ним*. Сказанное проиллюстрировано примером 19.1. Имейте в виду, что этот пример следует компоновать в проекте под именем ProgCSharp. Когда метод Main() создает объект Shape, он ищет файл *ProgCSharp.exe*.

Пример 19.1. Маршалинг через границы домена приложения

```
using System;
using System.Runtime.Remoting;

using System.Reflection;

namespace ProgCSharp
{


   // для маршалинга по ссылке,
   // закомментируйте атрибут и снимите комментарий с базового класса
   [Serializable]
   public class Point  // : MarshalByRefObject
   {
      public Point (int x, int y)
      {
         Console.WriteLine( "[{0}] {1}",
            System.AppDomain.CurrentDomain.FriendlyName,
            "Point constructor");

         this.x = x;
         this.y = y;
      }


      public int X
      {
         get
         {
            Console.WriteLine( "[{0}] {1}",
               System.AppDomain.CurrentDomain.FriendlyName,
               "Point x.get");

            return this.x;
         }

         set
         {
```

```
            Console.WriteLine( "[{0}] {1}",
               System.AppDomain.CurrentDomain.FriendlyName,
               "Point x.set");
            this.x = value;
         }
      }

      public int Y
      {
         get
         {
            Console.WriteLine( "[{0}] {1}",
               System.AppDomain.CurrentDomain.FriendlyName,
               "Point y.get");
            return this.y;
         }

         set
         {
            Console.WriteLine( "[{0}] {1}",
               System.AppDomain.CurrentDomain.FriendlyName,
               "Point y.set");
            this.y = value;
         }
      }

      private int x;
      private int y;
   }

   // класс Shape маршалируется по ссылке
   public class Shape : MarshalByRefObject
   {
      public Shape(int upperLeftX, int upperLeftY)
      {
         Console.WriteLine( "[{0}] {1}",
            System.AppDomain.CurrentDomain.FriendlyName,
            "Shape constructor");

         upperLeft = new Point(upperLeftX, upperLeftY);
      }
      public Point GetUpperLeft(  )
      {
         return upperLeft;
      }

      public void ShowUpperLeft(  )
      {
         Console.WriteLine( "[{0}] Upper left: {1},{2}",
            System.AppDomain.CurrentDomain.FriendlyName,
            upperLeft.X, upperLeft.Y);
      }

      private Point upperLeft;
```

```
    }
    public class Tester
    {
       public static void Main(  )
       {

          Console.WriteLine( "[{0}] {1}",
             System.AppDomain.CurrentDomain.FriendlyName,
             "Entered Main");

          // создать новый домен приложения
          AppDomain ad2 =
             AppDomain.CreateDomain("Shape Domain");

          //  Assembly a = Assembly.LoadFrom("ProgCSharp.exe");
          //  Object theShape = a.CreateInstance("Shape");
          // создать экземпляр объекта Shape
          ObjectHandle oh = ad2.CreateInstance(
             "ProgCSharp",
             "ProgCSharp.Shape", false,
             System.Reflection.BindingFlags.CreateInstance,
             null, new object[] {3, 5},
             null, null, null );

          Shape s1 = (Shape) oh.Unwrap(  );

          s1.ShowUpperLeft(  );     // запросить у объекта его положение

          // получить локальную копию заместителя?
          Point localPoint = s1.GetUpperLeft(  );

          // присвоить новые значения
          localPoint.X = 500;
          localPoint.Y = 600;

          // вывести значение локального объекта Point
          Console.WriteLine( "[{0}] localPoint: {1}, {2}",
             System.AppDomain.CurrentDomain.FriendlyName,
             localPoint.X, localPoint.Y);

          s1.ShowUpperLeft(  );     // показать значение еще раз
       }
    }
}
```

Вывод:

```
[Programming CSharp.exe] Entered Main
[Shape Domain] Shape constructor
[Shape Domain] Point constructor
[Shape Domain] Point x.get
[Shape Domain] Point y.get
[Shape Domain] Upper left: 3,5
[Programming CSharp.exe] Point x.set
[Programming CSharp.exe] Point y.set
[Programming CSharp.exe] Point x.get
```

```
[Programming CSharp.exe] Point y.get
[Programming CSharp.exe] localPoint: 500, 600
[Shape Domain] Point x.get
[Shape Domain] Point y.get
[Shape Domain] Upper left: 3,5
```

Внимательно изучите эту программу, а еще лучше – выполните ее в отладчике в пошаговом режиме. Из выведенного ею текста следует, что конструкторы Shape() и Point() работают в домене Shape Domain, как и программа, изменяющая значения свойств объекта Point в классе Shape.

Свойства этого объекта устанавливаются в первом домене приложения, где локальному экземпляру объекта присваиваются значения 500 и 600. Однако Point передается по значению, то есть фактически изменяется *копия* объекта Point. Если попросить объект Shape показать переменную upperLeft, будет видно, что она осталась прежней.

Чтобы завершить эксперимент, закомментируем атрибут в объявлении Point и снимем комментарий с базового класса:

```
// [serializable]
public class Point   : MarshalByRefObject
```

Теперь снова запустим программу. Выводимый текст изменится:

```
[Programming CSharp.exe] Entered Main
[Shape Domain] Shape constructor
[Shape Domain] Point constructor
[Shape Domain] Point x.get
[Shape Domain] Point y.get
[Shape Domain] Upper left: 3,5
[Shape Domain] Point x.set
[Shape Domain] Point y.set
[Shape Domain] Point x.get
[Shape Domain] Point y.get
[Programming CSharp.exe] localPoint: 500, 600
[Shape Domain] Point x.get
[Shape Domain] Point y.get
[Shape Domain] Upper left: 500,600
```

На этот раз создается заместитель объекта Point, и через него устанавливаются свойства исходной переменной типа Point. Таким образом, изменения отражаются в объекте Shape.

Контекст

Домены приложений делятся на *контексты*. Контексты можно представлять себе в виде границ, внутри которых объекты подчиняются единым правилам. В число этих правил входят синхронизация транзакций (см. главу 20) и некоторые другие.

Контекстно-связанные и контекстно-свободные объекты

Объекты могут быть либо *контекстно-связанными*, либо *контекстно-свободными*. В первом случае они существуют в контексте, и для взаимодействия с ними сообщение должно маршалироваться. Во втором – объекты работают в контексте вызвавшего объекта. Иными словами, их методы выполняются в контексте объекта, вызвавшего метод, и маршалинг не требуется.

Предположим, объект A взаимодействует с базой данных и поэтому поддерживает транзакции. Это создает контекст. Все вызовы методов объекта A происходят в контексте защиты, которую установила транзакция. Объект A может отменить транзакцию, и все действия, предпринятые с момента открытия транзакции, будут отменены.

Далее предположим, что имеется объект B и он контекстно свободен. Пусть объект A передаст базе данных ссылку на объект B и затем вызовет методы объекта B. Еще одно предположение: объекты A и B находятся в отношениях обратного вызова, то есть объект B, вызванный из A, проделает некоторую работу и вызовет A для передачи результатов. Поскольку B является контекстно свободным, его метод работает в контексте вызывающего объекта, и он будет находиться в границах защиты транзакции объекта A. Если A отменит транзакцию, изменения, вносимые объектом B в базу данных, тоже будут отменены, так как методы объекта B выполняются в контексте вызывающего объекта. Пока все хорошо.

Что лучше, контекстная связанность или контекстная свобода объекта B? В рассмотренном случае объект B прекрасно работал, будучи свободным. Предположим, существует еще один класс, C. Он не совершает транзакций и вызывает метод объекта B, вносящий изменения в базу данных. Объект A пытается отменить эти действия, но, к его сожалению, работа, выполненная объектом B для объекта C, выполнялась в контексте C, который не поддерживает транзакции. Эта работа не может быть отменена.

Если бы объект B был помечен как контекстно-связанный, то при его создании классом A он унаследовал бы контекст объекта A. В таком случае после вызова метода объекта B объектом C объекту B пришлось бы пересекать границы контекста. Но тогда метод, выполненный объектом B, будет оставаться в контексте транзакции объекта A. Это уже лучше.

Описанная схема работает при условии контекстной связанности объекта B, пока у него нет атрибутов. Конечно, объект B может иметь собственные атрибуты контекста и в соответствии с ними находиться вне контекста объекта A. Например, объект B может иметь атрибут транзакции `RequiresNew`. В таком случае при создании объекта B он получает новый контекст и по этой причине не может находиться в контексте A.

Иными словами, если объект A отменит транзакцию, он будет не в состоянии отменить работу метода B. Объект B может быть помечен перечислимым значением `RequiresNew`, например потому, что является аудиторской функцией. Когда объект A предпринимает действие с базой данных, он информирует объект B, а тот обновляет свой аудиторский журнал. Эта работа не должна быть отменена, даже если A отменит транзакцию. Объект B должен находиться в своем контексте транзакции и исправлять только свои ошибки, а не ошибки объекта A.

Таким образом, каждый объект стоит перед выбором из трех вариантов. Во-первых, он может быть контекстно-свободным. Контекстно-свободный объект работает в контексте вызывающего объекта. Во-вторых, объект может быть контекстно-связанным (что достигается выбором в качестве его базового класса `ContextBoundObject`), но без атрибутов. Тогда он работает в контексте своего создателя. В-третьих, он может быть контекстно-связанным и иметь контекстные атрибуты. Такой объект работает только в контексте, определенном атрибутами.

Решение, которое принимает разработчик, зависит от конкретных задач. Если объект представляет нехитрый калькулятор, которому не нужны синхронизация, транзакции и поддержка контекста, его следует сделать контекстно-свободным, так эффективнее. Если объекту понадобится контекст объекта, создавшего его, лучше объявить его контекстно-связанным без атрибутов. И наконец, объект с собственными контекстными требованиями должен иметь соответствующие атрибуты.

Маршалинг через границу контекста

При обращении к контекстно-свободным объектам из одного домена приложения заместители не нужны. Когда объект из одного контекста обращается к контекстно-связанному объекту из другого контекста, он делает это через заместитель, и в таком случае проводятся две контекстные политики. Именно в этом смысле говорят о границах контекста; политика проводится на границе, разделяющей контексты.

Например, если контекстно-связанный объект помечен атрибутом `System.EnterpriseServices.Synchronization`, это означает, что система должна управлять его синхронизацией. Все объекты вне этого контекста должны проходить через его границу, чтобы обратиться к одному из объектов в контексте. В этот момент и проводится политика синхронизации.

Строго говоря, пометка двух классов атрибутом `Synchronization` не гарантирует, что они окажутся в одном контексте. В момент активизации объекта каждый атрибут опрашивается на предмет того, соответствует ли ему текущий контекст. Если два объекта предназначены для синхронизации, но один помещен в пул, то они принудительно помещаются в разные контексты.

Объекты по-разному маршалируются через границу контекста, в зависимости от способа их создания:

- Обычные объекты вообще не маршалируются, а в пределах одного домена приложения они контекстно свободны.
- Объекты, помеченные атрибутом `Serializable`, являются контекстно-свободными и маршалируются по значению.
- Объекты, представляющие собой потомков класса `MarshalByRefObject`, являются контекстно-свободными и маршалируются по ссылке.
- Объекты, представляющие собой потомков класса `ContextBoundObject`, маршалируются по ссылке при передаче их как через границы доменов, так и через границы контекстов.

Удаленные объекты

Объекты могут маршалироваться не только для передачи их через границы контекста или домена приложения, но и через границы процесса и даже компьютера. Объект, для которого выполняется маршалинг (либо по значению, либо через посредника) для передачи через границы процесса или компьютера, называют *удаленным* (*remote*).

Виды серверных объектов

Существует два вида серверных объектов, отдаление которых поддерживается платформой .NET: *известные* (*well-known*) и *активизированные клиентом* (*client-activated*). Соединение с первыми устанавливается всякий раз, когда клиент посылает сообщение. С известным объектом постоянное соединение не устанавливается, в отличие от активизированных клиентом объектов.

Известные объекты, в свою очередь, делятся на два вида: *одиночные* (*singleton*) и *одноразовые* (*single-call*, то есть на один вызов). В случае одиночного известного объекта все сообщения от всех клиентов направляются одному объекту, работающему на сервере. Этот объект создается при запуске сервера и находится на нем для предоставления сервиса любому клиенту, который сможет к нему обратиться. Известные серверные объекты должны иметь конструкторы без параметров.

В случае с одноразовыми известными объектами каждое новое сообщение от клиента обрабатывается новым объектом. Это очень удобно на серверных фермах, где последовательные сообщения от одного клиента могут быть обработаны разными компьютерами, в зависимости от их загруженности.

Объекты, активизированные клиентом, обычно используются программистами, создающими для одного заказчика и программный продукт, и специализированный сервер, оказывающий ему услуги. В таком случае клиент и сервер устанавливают соединение, которое они поддерживают, пока сервер полностью не обслужит клиента.

Указание сервера с помощью интерфейса

Лучший способ разобраться в реализации доступа к объектам, реализованным в другом процессе, – разобрать конкретный пример. Создадим простой класс-калькулятор с четырьмя функциями, аналогичный созданному ранее для демонстрации работы веб-службы (см. главу 16). Этот класс реализует интерфейс, представленный в примере 19.2.

Пример 19.2. Интерфейс Калькулятора

```
namespace Programming_CSharp
{
    using System;

    public interface ICalc
    {
        double Add(double x, double y);
        double Sub(double x, double y);
        double Mult(double x, double y);
        double Div(double x, double y);
    }
}
```

Сохраните этот файл под именем *Icalc.cs* и откомпилируйте его в файл *ICalc.dll*. Чтобы создать файл и откомпилировать его в среде Visual Studio .NET, откройте новый проект типа Class Library, замените заготовку приложения, созданную средой разработки, на текст из примера 19.2 и выберите команду Build → Build в меню Visual Studio .NET. В качестве альтернативы можно ввести исходный текст программы в Блокноте и откомпилировать исходный файл, введя в командную строку следующую команду:

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

Реализация сервера через интерфейс обладает массой достоинств. Если реализовать калькулятор в виде класса, клиент будет вынужден указать при компоновке файл библиотеки динамической компоновки сервера, чтобы иметь возможность объявлять его экземпляры. Это существенно уменьшит преимущества, полученные в результате выделения объекта в сервер, поскольку изменения в классе на сервере повлекут за собой изменения на компьютере клиента. Иными словами, клиент и сервер будут тесно связаны друг с другом. Интерфейсы позволяют развести их. Впоследствии можно будет вносить изменения в реализацию сервера, но пока сервер выполняет договор, установленный интерфейсом, никакие изменения на компьютере клиента не потребуются.

Создание сервера

Чтобы создать сервер, используемый в рассматриваемом примере, создадим файл *CalcServer.cs* в новом проекте типа C# Console Applicati-

on (не забудьте включить в проект ссылку на *ICalc.dll*) и откомпилируем его, выбрав команду Build → Build в меню Visual Studio .NET. Если читатель предпочитает создавать код в Блокноте, файл *CalcServer.cs* следует откомпилировать, введя в командную строку команду:

```
csc /t:exe /r:ICalc.dll CalcServer.cs
```

Класс `Calculator` реализует интерфейс `ICalc`. Он является производным от класса `MarshalByRefObject`, что позволит предоставить клиентскому приложению заместитель калькулятора:

```
public class Calculator : MarshalByRefObject, ICalc
```

Реализация содержит конструктор, четыре метода, реализующие арифметические действия и еще кое-что.

В этом примере логика работы сервера помещена в метод `Main()` в файле *CalcServer.cs*.

Сначала необходимо создать канал. В качестве транспортного протокола воспользуемся HTTP, потому что он проще, а постоянное соединение TCP/IP здесь не требуется. Платформа .NET предоставляет программисту тип `HTTPChannel`:

```
HTTPChannel chan = new HTTPChannel(65100);
```

Обратите внимание, что канал зарегистрирован на порту TCP/IP с номером 65100 (обсуждение номеров портов см. в главе 21).

Далее зарегистрируем канал с помощью класса CLR `ChannelServices`, для чего вызовем статический метод `RegisterChannel`:

```
ChannelServices.RegisterChannel(chan);
```

В этом операторе платформе .NET сообщается о том, что сервер будет оказывать услуги по протоколу HTTP на порту 65100, подобно тому как IIS делает это на порту 80. Поскольку при регистрации HTTP-канала не был указан форматизатор, вызовы методов будут пользоваться форматизатором SOAP по умолчанию.

Теперь все готово к тому, чтобы класс `RemotingConfiguration` зарегистрировал известный серверный объект. Для этого программист должен передать ему тип регистрируемого объекта и *конечную точку* (*endpoint*). Конечная точка – это имя, которое класс `RemotingConfiguration` свяжет с предоставленным типом. Оно будет завершать адрес. В то время как IP-адрес идентифицирует компьютер, а номер порта идентифицирует канал, конечная точка идентифицирует приложение, предоставляющее сервис. Чтобы получить тип объекта, вызовем статический метод `GetType()` класса `Type` и передадим методу полное имя объекта:

```
Type calcType = Type.GetType("Programming_CSharp.Calculator");
```

Кроме типа и конечной точки при регистрации объекта следует указать перечислимое значение, уточняющее, одиночный он или одноразовый (то есть Singleton или SingleCall):

```
RemotingConfiguration.RegisterWellKnownServiceType
   ( calcType, "theEndPoint",WellKnownObjectMode.Singleton );
```

Вызов метода RegisterWellKnownServiceType() не пересылает данные, что называется, по проводам. Он просто создает заместитель объекта с помощью отражения.

Сервер Calculator создан. Его полный исходный текст содержится в примере 19.3.

Пример 19.3. Сервер Calculator

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

namespace Programming_CSharp
{
   // реализовать класс Calculator
   public class Calculator : MarshalByRefObject, ICalc
   {
      public Calculator()
      {
         Console.WriteLine("Calculator constructor");
      }
      // реализовать четыре функции
      public double Add(double x, double y)
      {
         Console.WriteLine("Add {0} + {1}", x, y);
         return x+y;
      }
      public double Sub(double x, double y)
      {
         Console.WriteLine("Sub {0} - {1}", x, y);
         return x-y;
      }
      public double Mult(double x, double y)
      {
         Console.WriteLine("Mult {0} * {1}", x, y);
         return x*y;
      }
      public double Div(double x, double y)
      {
         Console.WriteLine("Div {0} / {1}", x, y);
         return x/y;
      }
   }
```

```
    public class ServerTest
    {
       public static void Main()
       {
          // создать и зарегистрировать канал
          HttpChannel chan = new HttpChannel(65100);
          ChannelServices.RegisterChannel(chan);

          Type calcType =
             Type.GetType("Programming_CSharp.Calculator");

         // зарегистрировать известный тип и указать серверу на необходимость
          // связать его с конечной точкой theEndPoint
          RemotingConfiguration.RegisterWellKnownServiceType
             ( calcType,
               "theEndPoint",
                WellKnownObjectMode.Singleton );

          //  "Те тоже служат, кто лишь стоит и ждет." (Мильтон)
          Console.WriteLine("Press [enter] to exit...");
          Console.ReadLine();
       }
    }
}
```

Если запустить эту программу, она выведет скромное сообщение:

```
Press [enter] to exit...
```

и будет ждать, пока клиент не запросит услугу.

Создание клиента

Клиент тоже должен зарегистрировать канал, но поскольку по этому каналу не будет передаваться информация, можно воспользоваться каналом 0:

```
HTTPChannel chan = new HTTPChannel(0);
ChannelServices.RegisterChannel(chan);
```

Теперь клиенту надо лишь соединиться с удаленным сервером, передав объект `Type`, представляющий тип требуемого объекта (в данном случае интерфейс `ICalc`) и URI-идентификатор (URI расшифровывается как Uniform Resource Identifier, универсальный идентификатор ресурса):

```
MarshalByRefObject obj =
    RemotingServices.Connect
        (typeof(Programming_CSharp.ICalc),
        "http://localhost:65100/theEndPoint");
```

В этом примере предполагается, что сервер работает на локальном компьютере, поэтому URI состоит из адреса *http://localhost*, за которым следует номер порта сервера 65100, а за ним – конечная точка, объявленная на сервере, `theEndPoint`.

Удаленная служба должна возвратить объект, представляющий запрошенный интерфейс. После явного преобразования типа этого объекта он будет готов к употреблению. Поскольку установка удаленного соединения с объектом не имеет стопроцентной гарантии (в сети возможны неполадки, хост-компьютер может оказаться недоступным и т. д.), обращаться к объекту следует из блока `try`:

```
try
{
    Programming_CSharp.ICalc calc =
        obj as Programming_CSharp.ICalc;

    double sum = calc.Add(3,4);
```

В результате этих действий создан заместитель `Calculator`, работающий на сервере, но доступный клиенту за границами процесса и, если потребуется, за границами компьютера. Код клиента представлен в примере 19.4 (для его компиляции, так же как и в случае с *CalcServer.cs*, необходимо включить в проект ссылку на *ICalc.dll*).

Пример 19.4. Клиент класса Calculator

```
namespace Programming_CSharp
{
   using System;
   using System.Runtime.Remoting;
   using System.Runtime.Remoting.Channels;
   using System.Runtime.Remoting.Channels.Http;

   public class CalcClient
   {
      public static void Main()
      {

         int[] myIntArray = new int[3];

         Console.WriteLine("Watson, come here I need you...");

         // создать HTTP-канал и зарегистрировать его,
         // воспользоваться портом 0, чтобы показать,
         // что канал не используется
         HttpChannel chan = new HttpChannel(0);
         ChannelServices.RegisterChannel(chan);

         // получить объект по HTTP-каналу
         MarshalByRefObject obj =
          (MarshalByRefObject)  RemotingServices.Connect
            (typeof(Programming_CSharp.ICalc),
            "http://localhost:65100/theEndPoint");

         try
         {
            // привести тип объекта к типу интерфейса
            Programming_CSharp.ICalc calc =
               obj as Programming_CSharp.ICalc;
```

```
            // воспользоваться интерфейсом для вызова методов
            double sum = calc.Add(3.0,4.0);
            double difference = calc.Sub(3,4);
            double product = calc.Mult(3,4);
            double quotient = calc.Div(3,4);

            // вывести результаты
            Console.WriteLine("3+4 = {0}", sum);
            Console.WriteLine("3-4 = {0}", difference);
            Console.WriteLine("3*4 = {0}", product);
            Console.WriteLine("3/4 = {0}", quotient);
         }
         catch( System.Exception ex )
         {
            Console.WriteLine("Exception caught: ");
            Console.WriteLine(ex.Message);
         }
      }
   }
}
```

Вывод клиента:

```
Watson, come here I need you...
3+4 = 7

3-4 = -1
3*4 = 12
3/4 = 0.75
```

Вывод сервера:

```
Calculator constructor
Press [enter] to exit...
Add 3 + 4
Sub 3 - 4
Mult 3 * 4
Div 3 / 4
```

После запуска сервер ждет, пока пользователь не нажмет клавишу <Enter>, сигнализируя о конце работы. Запущенный клиент выводит на экран сообщение, после чего вызывает каждую из четырех функций. Сервер выводит сообщение после каждого вызова, а затем результаты выводятся клиентом.

На самом деле ничего сложного здесь нет. Некоторая программа работает на сервере и оказывает услуги клиенту.

Использование типа SingleCall

Чтобы лучше понять разницу между типами `SingleCall` и `Singleton`, изменим одну строчку в методе `Main()` сервера `Calculator`. Иными словами, строку:

```
RemotingServices. RegisterWellKnownServiceType
```

```
    ( "CalcServerApp","Programming_CSharp.Calculator",
        "theEndPoint",WellKnownObjectMode.Singleton );
```

заменим строкой:

```
RemotingServices. RegisterWellKnownServiceType
    ( "CalcServerApp","Programming_CSharp.Calculator",
        "theEndPoint",WellKnownObjectMode.SingleCall );
```

Теперь, судя по выводимой информации, для обработки каждого запроса создается новый объект:

```
Calculator constructor
Press [enter] to exit...
Calculator constructor
Add 3 + 4
Calculator constructor
Sub 3 - 4
Calculator constructor
Mult 3 * 4
Calculator constructor
Div 3 / 4
```

Метод RegisterWellKnownServiceType()

Что происходит, когда на сервере вызывается метод RegisterWellKnownServiceType()? Вспомним, что для класса Calculator был создан объект Type:

```
Type.GetType("Programming_CSharp.Calculator");
```

После этого был вызван метод RegisterWellKnownServiceType() с тремя аргументами: объектом типа Type, конечной точкой и перечислимым значением Singleton. Это послужило сигналом среде CLR, что нужно создать экземпляр класса Calculator и связать его с конечной точкой.

Можно проделать то же самое вручную. Для этого читатель должен изменить пример 19.3 таким образом, чтобы метод Main() создавал объект класса Calculator и передавал его методу Marshal() класса RemotingServices вместе с конечной точкой, с которой нужно связать этот объект класса Calculator. Измененный метод Main() приведен в примере 19.5. Нетрудно убедиться, что выводимый им текст идентичен тексту, выводимому программой из примера 19.3.

Пример 19.5. Создание объекта класса Calculator и связывание его с конечной точкой вручную

```
public static void Main()
{
   // создать канал и зарегистрировать его
   HttpChannel chan = new HttpChannel(65100);
   ChannelServices.RegisterChannel(chan);
```

```
        // создать объект класса вручную и вызвать метод Marshal()
        Calculator calculator = new Calculator();
        RemotingServices.Marshal(calculator,"theEndPoint");

        //  "Те тоже служат, кто лишь стоит и ждет." (Мильтон)
        Console.WriteLine("Press [enter] to exit...");
        Console.ReadLine();
    }
```

В результате выполнения этой программы создается объект-калькулятор, заместитель которого связывается с указанной конечной точкой.

Конечные точки

Что происходит во время регистрации конечной точки? Очевидно, сервер связывает ее с только что созданным объектом, а после установки соединения с клиентом эта конечная точка служит индексом в таблице, чтобы сервер мог предоставить объекту (в данном случае объекту `Calculator`) заместитель.

Существует альтернатива указанию конечной точки, с которой мог бы связаться клиент. Программист может записать в файл всю информацию об объекте-калькуляторе и физически передать этот файл клиенту. Например, можно послать файл по электронной почте, чтобы пользователь загрузил его на локальном компьютере.

Клиент прочтет посланную информацию об объекте и воспроизведет заместитель, которым сможет воспользоваться для доступа к калькулятору на сервере! (Следующий пример был предложен автору Майком Вудрингом (Mike Woodring) из DevelopMentor. Майк использовал аналогичный пример для доказательства идеи, что конечная точка является не более чем удобным способом удаленного доступа к маршалируемому объекту.)

Чтобы увидеть, как можно вызвать объект, не зная конечной точки, еще раз внесем изменения в метод `Main()` из примера 19.3. На этот раз передадим методу `Marshal()` только объект, без конечной точки:

```
ObjRef objRef = RemotingServices.Marshal(calculator)
```

Метод возвратит объект типа `ObjRef`, который будет содержать всю информацию, необходимую для активизации удаленного объекта и общения с ним. Если конечная точка указана, сервер создает таблицу, которая связывает конечную точку с объектом типа `ObjRef`, чтобы сервер мог создать заместитель, когда клиент попросит об этом. Объект `objRef` содержит всю информацию, необходимую клиенту для самостоятельного создания заместителя, кроме того, сам объект `objRef` может хранить свое состояние в сериализованном виде.

Откроем файловый поток для записи в новый файл и создадим новый форматизатор SOAP. Объект `objRef` может записать свое состояние в этот файл, вызвав метод `Serialize()` форматизатора SOAP и передав

ему в качестве аргументов файловый поток и объект objRef, возвращенный методом Marshal(). Готово! В файле на диске содержится вся необходимая информация, позволяющая создать заместитель. Полный текст новой версии метода Main() приведен в примере 19.6. Также потребуется добавить два оператора using в файл *CalcServer.cs*:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
```

Пример 19.6. Составление объекта без известной конечной точки

```
public static void Main()
{
   // создать канал и зарегистрировать его
   HttpChannel chan = new HttpChannel(65100);
   ChannelServices.RegisterChannel(chan);
   // создать объект вручную и вызвать метод Marshal()
   Calculator calculator = new Calculator();

   ObjRef objRef = RemotingServices.Marshal(calculator);

   FileStream fileStream =
      new FileStream("calculatorSoap.txt",FileMode.Create);
   SoapFormatter soapFormatter = new SoapFormatter();

   soapFormatter.Serialize(fileStream,objRef);
   fileStream.Close();

   //  "Те тоже служат, кто лишь стоит и ждет." (Мильтон)
   Console.WriteLine(
   "Exported to CalculatorSoap.txt. Press ENTER to exit...");
   Console.ReadLine();
}
```

Если теперь запустить сервер, он запишет на диск файл *calculatorSoap.txt*. Потом сервер перейдет в состояние ожидания, в котором может пробыть довольно долго, пока с ним не соединится клиент.

Полученный файл можно перенести на компьютер клиента и восстановить там объект. С этой целью снова создадим и зарегистрируем канал. Впрочем, на этот раз поток fileStream откроем для файла, скопированного с сервера:

```
FileStream fileStream = new FileStream ("calculatorSoap.txt",
FileMode.Open);
```

Затем создадим объект форматизатора SoapFormatter и вызовем его метод Deserialize(), передав ему имя файла. Метод возвратит объект ICalc:

```
SoapFormatter soapFormatter = new SoapFormatter ();
try
{
   ICalc calc = (ICalc) soapFormatter.Deserialize.(fileStream)
```

Теперь можно свободно вызывать методы на сервере, пользуясь ссылкой ICalc, действующей как заместитель объекта-калькулятора, работающего на сервере и описанного в файле *calculatorSoap.txt*. Полный текст нового метода Main() клиента приведен в примере 19.7. Как и в прошлый раз, потребуется добавить два метода using:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
```

Пример 19.7. Новая версия метода Main( ) клиента из примера 19.4

```
public static void Main(  )
{

   int[] myIntArray = new int[3];

   Console.WriteLine("Watson, come here I need you...");

   // создать HTTP-канал и зарегистрировать его,
   // воспользоваться портом 0, чтобы показать,
   // что канал не слушается
   HttpChannel chan = new HttpChannel(0);
   ChannelServices.RegisterChannel(chan);
   FileStream fileStream = new FileStream ("calculatorSoap.txt",
FileMode.Open);
   SoapFormatter soapFormatter = new SoapFormatter ();

   try
   {
      ICalc calc= (ICalc) soapFormatter.Deserialize (fileStream);

      // использовать интерфейс для вызова методов
      double sum = calc.Add(3.0,4.0);
      double difference = calc.Sub(3,4);
      double product = calc.Mult(3,4);
      double quotient = calc.Div(3,4);

      // вывести результаты
      Console.WriteLine("3+4 = {0}", sum);
      Console.WriteLine("3-4 = {0}", difference);
      Console.WriteLine("3*4 = {0}", product);
      Console.WriteLine("3/4 = {0}", quotient);
   }
   catch( System.Exception ex )
   {
      Console.WriteLine("Exception caught: ");
      Console.WriteLine(ex.Message);
   }
}
```

Клиент, начиная свою работу, читает файл с диска и демаршалирует заместитель. Это будет операция, обратная маршализации и сериализации на сервере. Создав заместитель, клиент может вызывать методы калькулятора, работающего на сервере.

20

Потоки и синхронизация

Потоки – это облегченные версии процессов, позволяющие организовать многозадачность в рамках одного приложения. Пространство имен `System.Threading` предоставляет программисту богатый выбор классов и интерфейсов, позволяющих создавать многопоточные программы. Впрочем, большинству программистов вряд ли когда-нибудь придется управлять потоками явно, потому что среда CLR почти всю поддержку потоков абстрагирует в классы, которые существенно упрощают многие задачи управления потоками. Так, в главе 21 будет продемонстрировано, как организовать многопоточный ввод/вывод данных и избежать при этом управления потоками вручную.

В первой части этой главы показано, как создавать потоки, управлять ими и уничтожать их. Даже если читатель не будет создавать потоки явно, он должен будет писать программы, способные корректно работать с несколькими потоками, если они выполняются в многопоточной среде. Эта тема становится особенно актуальной, если читатель разрабатывает компоненты, которые используются разработчиками в программах, поддерживающих многопоточность. В частности, это важно при создании веб-служб. Хотя веб-службы (см. главу 16) имеют много общего с обычными приложениями, они работают на сервере и часто не имеют пользовательского интерфейса, что заставляет разработчиков задумываться о таких сторонах серверного программирования, как эффективность и многопоточность.

Вторая часть главы посвящена синхронизации. При наличии ограниченного ресурса приходится допускать к нему не более одного потока одновременно. Классической аналогией является туалет в самолете. Зайти в него может только один человек. Для реализации этого требования существует замок на двери. Захотев попасть в туалет, пассажир

дергает ручку двери. Если дверь заперта, он либо уходит, либо терпеливо ждет (возможно, в очереди), пока ресурс не освободится. Когда это произойдет, первый из очереди заходит в туалет и запирает дверь на замок.

Время от времени различные потоки пытаются обратиться к какому-либо ресурсу программы, например к файлу. Если по каким-то соображениям нельзя допустить одновременного обращения к ресурсу нескольких потоков, программист блокирует ресурс, позволяет потоку обратиться к нему, а затем снимает блокировку. Программная блокировка может быть довольно сложной, если, например, требуется справедливое распределение ресурсов.

Потоки

В типичном случае потоки создаются, когда программа хочет делать два дела одновременно. Предположим, программа вычисляет значение числа π (3,141592653589...) с точностью до 10-миллиардного знака. Процессор начнет вычисления, но пока он этим занят, на экран ничего выводиться не будет. Поскольку выполнение такого рода задачи продлится несколько миллионов лет, было бы разумно попросить процессор время от времени сообщать о ходе вычисления. Кроме того, можно добавить в интерфейс кнопку «Стоп», посредством которой пользователь сможет прервать эту операцию в любой момент. Чтобы программа могла обработать нажатие кнопки, нужен второй поток.

Отделение (apartment) – это логический контейнер внутри процесса, который содержит объекты, имеющие одинаковые требования на доступ к потокам. Все объекты в отделении могут принимать вызовы методов от любого объекта в любом потоке из этого отделения. Платформа .NET Framework не пользуется отделениями, и ответственность за безопасность одновременной работы с несколькими потоками лежит на управляемых объектах (объектах, созданных в среде CLR). Единственным исключением является ситуация, в которой управляемый код общается к COM-объектам. Взаимодействие с COM обсуждается в главе 22.

Другой распространенной ситуацией, в которой используются потоки, является ожидание какого-либо события, например пользовательского ввода, чтения файла или получения данных через сеть. Освобождение процессора и переключение его на другую работу на время ожидания (пусть он пока вычислит еще 10 000 знаков числа π) будет разумным решением, а программа не будет казаться пользователю такой медлительной.

С другой стороны, необходимо отметить, что в некоторых случаях многопоточность может фактически замедлить работу программы.

Предположим, что кроме вычисления числа π необходимо вычислять числа Фибоначчи (1, 1, 2, 3, 5, 8, 13, 21 ...). На компьютере с несколькими процессами лучше (и быстрее) решать каждую задачу в отдельном потоке. Если же у компьютера один процессор (что бывает в подавляющем большинстве случаев), то вычисление этих значений в нескольких потоках определенно будет протекать *медленнее*, чем последовательное выполнение сначала одной задачи, затем другой в одном потоке. Дело в том, что переключение процессора с одного потока на другой требует накладных расходов.

Запуск потоков

Простейший способ запустить поток – создать новый объект класса `Thread`. Конструктор `Thread()` имеет единственный аргумент типа `delegate`. Специально для этой цели в среде CLR создан класс делегата `ThreadStart`, который будет указывать на переданный метод. Это позволяет программисту создать поток и велеть ему выполнить метод при своем запуске. Делегат `ThreadStart` объявляется следующим образом:

```
public delegate void ThreadStart();
```

Как видно из объявления, метод, закрепляемый за делегатом, не должен иметь параметров и не должен возвращать значения. Например, новый поток можно создать так:

```
Thread myThread = new Thread( new ThreadStart(myFunc) );
```

Метод `myFunc` не должен иметь параметров и не должен возвращать значения.

Создадим два рабочих потока, один из которых увеличивает значение счетчика начиная с нуля:

```
public void Incrementer()
{
    for (int i =0;i<10;i++)
    {
        Console.WriteLine("Incrementer: {0}", i);
    }
}
```

а второй – уменьшает начиная с 10:

```
public void Decrementer()
{
    for (int i = 10;i>=0;i--)
    {
        Console.WriteLine("Decrementer: {0}", i);
    }
}
```

Чтобы запустить эти потоки, создадим два новых потока и проинициализируем каждый делегатом ThreadStart, который проинициализируем соответствующими функциями класса:

```
Thread t1 = new Thread( new ThreadStart(Incrementer) );
Thread t2 = new Thread( new ThreadStart(Decrementer) );
```

Создание объектов этих потоков еще не приводит к их запуску. Необходимо вызвать метод Start() объекта Thread:

```
t1.Start();
t2.Start();
```

Если не предпринять никаких дополнительных действий, поток будет остановлен, когда функция возвратит значение. Далее в этой главе будет показано, как останавливать поток до окончания работы функции.

В примере 20.1 приведен полный текст программы и выводимый ею текст. Оператор using для пространства имен System.Threading поставлен для того, чтобы компилятор мог получить доступ к классу Thread. Обратите внимание, что выводимая информация свидетельствует о переключении процессора с потока t1 на поток t2.

Пример 20.1. Использование потоков

```
namespace Programming_CSharp
{
   using System;
   using System.Threading;

   class Tester
   {
      static void Main()
      {
         // создать объект класса
         Tester t = new Tester();

         // выполнить вне статического метода Main()
         t.DoTest();

      }

      public void DoTest()
      {
         // создать поток для метода Incrementer() и
         // передать делегат ThreadStart
         // с адресом метода Incrementer()
         Thread t1 =
            new Thread(
               new ThreadStart(Incrementer) );
         // создать поток для метода Decrementer() и
         // передать делегат ThreadStart
```

```
            // с адресом метода Decrementer()
            Thread t2 =
               new Thread(
                  new ThreadStart(Decrementer) );

            // запустить потоки
            t1.Start();
            t2.Start();
         }

         // демонстрационная функция считает до 999
         public void Incrementer()
         {
            for (int i =0;i<1000;i++)
            {
               Console.WriteLine("Incrementer: {0}", i);
            }
         }

         // демонстрационная функция ведет обратный отсчет от 1000
         public void Decrementer()
         {
            for (int i = 1000;i>=0;i--)
            {
               Console.WriteLine("Decrementer: {0}", i);
            }
         }
      }
   }
```

Вывод (фрагмент):

```
Incrementer: 102
Incrementer: 103
Incrementer: 104
Incrementer: 105
Incrementer: 106
Decrementer: 1000
Decrementer: 999
Decrementer: 998
Decrementer: 997
```

Как видно из выводимого текста, процессор позволил первому потоку досчитать до 106. Затем второй поток начал обратный отсчет от 1000, но через некоторое время возобновился первый. Когда автор выполнил эту программу с достаточно большими числами, оказалось, что каждый поток успевает выполнить около 100 итераций прежде, чем процессор переключится на второй. Фактическое время, отводимое одному потоку, определяется диспетчером потоков и зависит от многих факторов, включая быстродействие процессора, запросы от других программ и т. д.

Объединение потоков

Когда программист велит одному потоку остановиться и ждать, пока второй поток не закончит свою работу, говорят, что программист объединяет два потока. Можно представить себе, будто первый поток «привязывается» к хвосту второго потока, и это их объединяет в один поток.

Чтобы присоединить поток t1 к потоку t2, следует написать:

```
t2.Join();
```

Если данный оператор выполнить в каком-нибудь методе потока t1, то этот поток остановится и будет ждать окончания потока t2. Например, можно приостановить поток, в котором выполняется метод Main(), и велеть ему подождать, пока закончатся все остальные потоки, чтобы после этого вывести заключительное сообщение. В следующем фрагменте программы предполагается, что создана коллекция потоков по имени myThreads. Все элементы перебираются в цикле, и текущий поток поочередно объединяется с каждым потоком из коллекции:

```
foreach (Thread myThread in myThreads)
{
    myThread.Join();
}

Console.WriteLine("Все мои потоки закончились.");
```

Заключительное сообщение «Все мои потоки закончились» будет выведено только после того, как все потоки завершат работу. В реальной программе был бы запущен целый ряд потоков, выполняющих некоторые задания (например печать, обновление информации на дисплее), а главный поток ждал бы, когда закончатся все вспомогательные.

Приостановка потоков

Иногда бывает необходимо приостановить поток на короткое время. Например, исполнение потока, выводящего на экран время, можно приостанавливать примерно на секунду перед тем, как он проверит системное время в очередной раз. Это позволит обновлять показания часов каждую секунду, не затрачивая при этом сотни миллионов машинных циклов.

Класс Thread обладает открытым статическим методом Sleep(), предусмотренным специально для приостановки выполнения потока. Этот метод является перегруженным, и одна из его версий принимает значение типа int, а другая – объект timeSpan. В обоих случаях аргумент содержит число миллисекунд, на которое приостанавливается поток, выраженное либо целым числом (например, 2000 соответствует двум секундам), либо объектом timeSpan.

Хотя объекты timeSpan измеряют время в *тиках* (внесистемная единица, равная 100 наносекундам), точность метода Sleep() – одна миллисекунда (1 000 000 наносекунд).

Чтобы приостановить поток на одну секунду, следует вызвать статический метод Sleep() класса Thread, приостанавливающий поток, в котором он вызван:

```
Thread.Sleep(1000);
```

Иногда программист приостанавливает поток лишь на одну миллисекунду. Это делается, чтобы сообщить диспетчеру потоков, что данный поток должен уступить место другому, даже если диспетчер готов выделить ему еще немного времени.

Если изменить пример 20.2 так, что после каждого вызова WriteLine() будет стоять метод Thread.Sleep(1), то выводимый программой текст изменится коренным образом:

```
for (int i =0;i<1000;i++)
{
   Console.WriteLine("Incrementer: {0}", i);
   Thread.Sleep(1);
}
```

Этого небольшого изменения достаточно, чтобы каждый поток возобновлялся после того, как другой выведет одно значение. Теперь вывод программы выглядит иначе:

```
Incrementer: 0
Incrementer: 1
Decrementer: 1000
Incrementer: 2
Decrementer: 999
Incrementer: 3
Decrementer: 998
Incrementer: 4
Decrementer: 997
Incrementer: 5
Decrementer: 996
Incrementer: 6
Decrementer: 995
```

Уничтожение потоков

Как правило, потоки уничтожаются по окончании их работы. Однако программист может отдать потоку приказ на самоуничтожение, вызвав метод Abort(). Этот метод вызывает исключение ThreadAbortException. Оно перехватывается потоком, что дает ему возможность освободить выделенные ему ресурсы (если таковые имеются).

```
catch (ThreadAbortException)
{
```

```
        Console.WriteLine("[{0}] Aborted! Cleaning up...",
            Thread.CurrentThread.Name);
    }
```

Поток обязан воспринять исключение ThreadAbortException как сигнал, что пора заканчивать работу, причем как можно скорее. Программист не столько уничтожает поток, сколько вежливо просит его самоуничтожиться.

Необходимость уничтожить поток может возникнуть в качестве реакции на какое-либо событие, например на нажатие кнопки Cancel. Если обработчик нажатия кнопки находится в потоке T1, а само событие возникнет в потоке T2, то обработчик вызовет метод Abort() потока T1:

```
T1.Abort();
```

Исключение будет вызвано в текущем методе потока T1, и поток T1 обработает его. Благодаря этому поток T1 получит возможность освободить ресурсы и элегантно закончить работу.

В примере 20.2 создаются три потока. Они сохраняются в массиве, элементы которого имеют тип Thread. До запуска этих потоков свойству IsBackground каждого из них присваивается значение true. Затем каждый поток запускается и получает имя Thread1, Thread2 и т. д. Запуск потока сопровождается выводом сообщения, после чего главный поток приостанавливается на 50 миллисекунд, а затем запускает следующий поток.

После запуска третьего потока главный поток ждет еще 50 миллисекунд и уничтожает второй поток методом Abort(). Затем главный поток объединяется с тремя запущенными потоками. В результате главный поток не сможет закончиться, пока не закончатся все остальные потоки. Дождавшись окончания их работы, он выведет сообщение «Все мои потоки закончились». Полный текст программы представлен в примере 20.2.

Пример 20.2. Прерывание потока

```
namespace Programming_CSharp
{
   using System;
   using System.Threading;

   class Tester
   {
      static void Main()
      {
         // создать объект данного класса
         Tester t = new Tester();

         // выполнить вне статического метода Main()
         t.DoTest();
      }
```

```
public void DoTest()
{
   // создать массив безымянных потоков
   Thread[] myThreads =
      {
         new Thread( new ThreadStart(Decrementer) ),
         new Thread( new ThreadStart(Incrementer) ),
         new Thread( new ThreadStart(Incrementer) )
      };

   // запустить каждый поток
   int ctr = 1;
   foreach (Thread myThread in myThreads)
   {
      myThread.IsBackground=true;
      myThread.Start();
      myThread.Name = "Thread" + ctr.ToString();
      ctr++;
      Console.WriteLine("Started thread {0}", myThread.Name);
      Thread.Sleep(50);
   }

   // запустив все потоки, дать второму команду
   // на самоуничтожение
   myThreads[1].Abort();

   // дождаться окончания работы всех потоков
   foreach (Thread myThread in myThreads)
   {
      myThread.Join();
   }

   // когда все потоки закончатся, вывести сообщение
   Console.WriteLine("Все мои потоки закончились.");
}

// демонстрационная функция ведет обратный отсчет от 1000
public void Decrementer()
{
   try
   {
      for (int i = 1000;i>=0;i--)
      {
         Console.WriteLine(
            "Thread {0}. Decrementer: {1}",
            Thread.CurrentThread.Name,
            i);
         Thread.Sleep(1);
      }
   }
   catch (ThreadAbortException)
   {
      Console.WriteLine(
```

```
                "Thread {0} aborted! Cleaning up...",
                Thread.CurrentThread.Name);
            }
            finally
            {
                Console.WriteLine(
                "Thread {0} Exiting. ",
                Thread.CurrentThread.Name);
            }
        }

        // демонстрационная функция считает до 999
        public void Incrementer()
        {
            try
            {
                for (int i =0;i<1000;i++)
                {
                    Console.WriteLine(
                        "Thread {0}. Incrementer: {1}",
                        Thread.CurrentThread.Name,
                        i);
                    Thread.Sleep(1);
                }
            }
            catch (ThreadAbortException)
            {
                Console.WriteLine(
                "Thread {0} aborted! Cleaning up...",
                Thread.CurrentThread.Name);
            }
            finally
            {
                Console.WriteLine(
                "Thread {0} Exiting. ",
                Thread.CurrentThread.Name);
            }
        }
    }
}
```

Вывод (отрывок):

```
Started thread Thread1
Thread Thread1. Decrementer: 1000
Thread Thread1. Decrementer: 999
Thread Thread1. Decrementer: 998
Started thread Thread2
Thread Thread1. Decrementer: 997
Thread Thread2. Incrementer: 0
Thread Thread1. Decrementer: 996
Thread Thread2. Incrementer: 1
Thread Thread1. Decrementer: 995
```

```
Thread Thread2. Incrementer: 2
Thread Thread1. Decrementer: 994
Thread Thread2. Incrementer: 3
Started thread Thread3
Thread Thread1. Decrementer: 993
Thread Thread2. Incrementer: 4
Thread Thread2. Incrementer: 5
Thread Thread1. Decrementer: 992
Thread Thread2. Incrementer: 6
Thread Thread1. Decrementer: 991
Thread Thread3. Incrementer: 0
Thread Thread2. Incrementer: 7
Thread Thread1. Decrementer: 990
Thread Thread3. Incrementer: 1
Thread Thread2 aborted! Cleaning up...
Thread Thread2 Exiting.
Thread Thread1. Decrementer: 989
Thread Thread3. Incrementer: 2
Thread Thread1. Decrementer: 988
Thread Thread3. Incrementer: 3
Thread Thread1. Decrementer: 987
Thread Thread3. Incrementer: 4
Thread Thread1. Decrementer: 986
Thread Thread3. Incrementer: 5
// ...
Thread Thread1. Decrementer: 1
Thread Thread3. Incrementer: 997
Thread Thread1. Decrementer: 0
Thread Thread3. Incrementer: 998
Thread Thread1 Exiting.
Thread Thread3. Incrementer: 999
Thread Thread3 Exiting.
Все мои потоки закончились.
```

Видно, как первый поток стартовал и отсчитал от 1000 до 998. После этого был запущен второй поток, и оба потока чередовались некоторое время, пока не появился третий. Впрочем, вскоре поток `Thread2` сообщает, что он прерван, а затем – что он заканчивает работу. Два оставшихся потока продолжают работу и заканчиваются естественным образом. Главный поток, который был объединен с ними, возобновляется лишь для того, чтобы вывести сообщение и закончить свою работу.

Синхронизация

Иногда требуется установить контроль над доступом к какому-либо ресурсу (например, к свойствам или методам объекта) так, чтобы в данный момент только один поток мог вносить изменения или другим образом использовать этот ресурс. Если применить аналогию, приведенную в начале главы, объект в этом случае подобен туалету в самоле-

те, а потоки – пассажирам, желающим в него попасть. Синхронизация обеспечивается блокировкой объекта, которая не позволяет второму потоку вторгаться, пока первый не освободит объект.

В этом разделе будут рассмотрены три механизма синхронизации, предоставляемые средой CLR: класс `Interlock`, оператор `lock` языка C# и класс `Monitor`. Но сперва необходимо создать имитатор совместно используемого ресурса, такого как файл или принтер. Этим имитатором будет целая переменная `counter`. Вместо того чтобы открывать файл или обращаться к принтеру, каждый из двух потоков будет просто увеличивать значение переменной.

Начнем с объявления переменной и инициализации ее нулем:

```
int counter = 0;
```

Изменим метод `Incrementer()` так, чтобы он увеличивал переменную `counter`:

```
public void Incrementer()
{
   try
   {
      while (counter < 1000)
      {
         int temp = counter;
         temp++; // инкрементировать

         // имитировать работу метода
         Thread.Sleep(1);

         // присвоить переменной counter
         // новое значение
         // и вывести результат
         counter = temp;
         Console.WriteLine(
            "Thread {0}. Incrementer: {1}",
            Thread.CurrentThread.Name,
            counter);
      }
   }
```

Идея состоит в том, чтобы имитировать работу, которая могла бы быть проделана над контролируемым ресурсом. Реальная программа открыла бы файл, обработала его содержимое и закрыла бы его. Здесь же значение переменной `counter` сохраняется во временной переменной, эта переменная увеличивается на 1, метод «спит» одну миллисекунду, после чего увеличенное значение записывается в переменную `counter`.

Описываемая ситуация чревата возникновением следующей типичной проблемы. Первый поток читает значение счетчика (0), присваивает его временной переменной и инкрементирует ее. Пока он делает это,

второй поток читает значение счетчика (все еще 0) и записывает его в свою временную переменную. Тут первый поток заканчивает работу, присваивает счетчику значение временной переменной (1) и выводит ее на экран. Второй поток делает то же самое. В результате вывод может выглядеть так: 1,1. На следующем шаге цикла повторяется то же самое. Итак, вместо последовательности значений 1,2,3,4 на экране появляется 1,1,2,2,3,3. Текст и выводимая этим примером информация содержатся в примере 20.3.

Пример 20.3. Имитация совместно используемого ресурса

```
namespace Programming_CSharp
{
   using System;
   using System.Threading;

   class Tester
   {
      private int counter = 0;

      static void Main()
      {
         // создать объект этого класса
         Tester t = new Tester();

         // выполнить вне статического метода Main()
         t.DoTest();
      }

      public void DoTest()
      {
         Thread t1 = new Thread( new ThreadStart(Incrementer) );
         t1.IsBackground=true;
         t1.Name = "ThreadOne";
         t1.Start();
         Console.WriteLine("Started thread {0}", t1.Name);

         Thread t2 = new Thread( new ThreadStart(Incrementer) );
         t2.IsBackground=true;
         t2.Name = "ThreadTwo";
         t2.Start();
         Console.WriteLine("Started thread {0}",
            t2.Name);
         t1.Join();
         t2.Join();

         // по окончании работы всех потоков вывести сообщение
         Console.WriteLine("All my threads are done.");
      }
      // демонстрационная функция считает до 1000
      public void Incrementer()
      {
         try
         {
```

```
            while (counter < 1000)
            {
               int temp = counter;
               temp++; // инкрементировать

               // имитировать работу метода
               Thread.Sleep(1);

               // присвоить переменной counter новое значение
               // и вывести результат
               counter = temp;
               Console.WriteLine(
                  "Thread {0}. Incrementer: {1}",
                  Thread.CurrentThread.Name,
                  counter);
            }
         }
         catch (ThreadAbortException)
         {
            Console.WriteLine(
               "Thread {0} aborted! Cleaning up...",
               Thread.CurrentThread.Name);
         }
         finally
         {
            Console.WriteLine(
               "Thread {0} Exiting. ",
               Thread.CurrentThread.Name);
         }
      }
   }
}
```

Вывод:

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadOne. Incrementer: 2
Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
Thread ThreadOne. Incrementer: 4
Thread ThreadTwo. Incrementer: 5
Thread ThreadOne. Incrementer: 5
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 6
```

Предположим, что два этих потока обращаются к базе данных, а не к переменной класса, а программа является частью системы управления складом в книжном магазине. Покупатель запрашивает, есть ли в наличии книга «Программирование на C#». Первый поток обнаруживает, что на складе есть один экземпляр, и покупатель выражает жела-

ние купить его. Поток приступает к выяснению номера кредитной карты и адреса покупателя.

Пока разворачиваются эти события, второй поток проверяет наличие книги на складе. Поскольку первый поток еще не обновил запись в базе данных, второй начинает процедуру продажи. Тем временем первый поток покончил с формальностями и уменьшил количество книг до нуля. Второй поток, находясь в неведении о действиях первого, тоже устанавливает нулевое значение в базе данных. К сожалению, один экземпляр книги оказался проданным дважды.

Как говорилось выше, доступ к объекту `counter` (или к записи в базе данных, файлу, принтеру и т. д.) должен быть синхронизирован.

Класс Interlocked

Среда CLR предоставляет программисту несколько механизмов синхронизации. В их число входят обычные инструменты, такие как критические секции (называемые *блокировками* в терминологии платформы .NET), а также более сложные средства, например класс `Monitor`. Все они обсуждаются далее в этой главе.

Инкрементирование/декрементирование значения является такой распространенной операцией в программировании, и она так часто требует синхронизации, что в языке C# предусмотрен специальный класс `Interlocked`. Этот класс обладает двумя методами, `Increment()` и `Decrement()`, которые не только увеличивают или уменьшают значение на 1, но и обеспечивают при этом синхронизацию.

Измените метод `Incrementer()` из примера 20.3 следующим образом:

```
public void Incrementer()
{
   try
   {
      while (counter < 1000)
      {
         temp=counter
         Interlocked.Increment(ref temp);

         // имитировать работу метода
         Thread.Sleep(1);

         // вывести результат
         counter=temp
         Console.WriteLine(
            "Thread {0}. Incrementer: {1}",
            Thread.CurrentThread.Name,
            counter);
      }
   }
}
```

Блоки catch и finally, а также остальную часть программы оставьте без изменений.

Метод Interlocked.Increment() имеет один аргумент, ссылку на объект типа int. Поскольку переменные целочисленных типов передаются по значению, перед аргументом следует поставить ключевое слово ref (см. главу 4).

Метод Increment() является перегруженным и может принять ссылку на объект типа long, если так удобнее программисту.

После внесения этих изменений доступ к переменной counter оказывается синхронизированным, а вывод – корректным:

Вывод (отрывок):

```
Started thread ThreadOne
Started thread ThreadTwo
Thread ThreadOne. Incrementer: 1
Thread ThreadTwo. Incrementer: 2
Thread ThreadOne. Incrementer: 3
Thread ThreadTwo. Incrementer: 4
Thread ThreadOne. Incrementer: 5
Thread ThreadTwo. Incrementer: 6
Thread ThreadOne. Incrementer: 7
Thread ThreadTwo. Incrementer: 8
Thread ThreadOne. Incrementer: 9
Thread ThreadTwo. Incrementer: 10
Thread ThreadOne. Incrementer: 11
Thread ThreadTwo. Incrementer: 12
Thread ThreadOne. Incrementer: 13
Thread ThreadTwo. Incrementer: 14
Thread ThreadOne. Incrementer: 15
Thread ThreadTwo. Incrementer: 16
Thread ThreadOne. Incrementer: 17
Thread ThreadTwo. Incrementer: 18
Thread ThreadOne. Incrementer: 19
Thread ThreadTwo. Incrementer: 20
```

Использование блокировок

Хотя объект Interlocked хорош при выполнении операций инкрементирования/декрементирования, иногда приходится управлять доступом и к другим объектам. Для этого нужен более общий механизм синхронизации, и в .NET он обеспечивается объектом Lock, представляющим собой блокировку.

Блокировка отмечает критическую секцию кода, обеспечивая синхронизацию доступа к указанному объекту, пока она включена. Правила обращения к объекту Lock заключаются в запросе на блокировку требуемого объекта и в выполнении оператора или блока операторов.

Язык C# предоставляет непосредственную поддержку блокировок через ключевое слово `lock`. Программисту достаточно передать объект ссылочного типа, и указать блок операторов после ключевого слова `lock`:

```
lock(выражение) блок-операторов
```

Например, можно еще раз внести изменения в метод `Incrementer()`, чтобы в нем был использован оператор блокировки:

```
public void Incrementer()
{
   try
   {
      while (counter < 1000)
      {
         lock (this)
         {
            int temp = counter;
            temp ++;
            Thread.Sleep(1);
            counter = temp;
         }

         // вывести результат
         Console.WriteLine(
            "Thread {0}. Incrementer: {1}",
            Thread.CurrentThread.Name,
            counter);
      }
   }
```

Блоки `catch` и `finally`, а также остальную часть программы оставим прежними.

Результат работы этого кода идентичен тому, что получился в предыдущем примере, где применялся класс `Interlocked`.

Использование мониторов

Объектов, рассмотренных выше, будет достаточно для решения большинства задач. Чтобы организовать более сложный контроль над доступом к ресурсам, потребуется *монитор* (*monitor*). Этот объект позволяет программисту решать, когда начинать, а когда заканчивать синхронизацию. Кроме того, он позволяет в одном участке программы реализовать ожидание того, когда освободится другой ее участок.

Монитор действует как «интеллектуальная» блокировка. Когда программист хочет начать синхронизацию, он вызывает метод `Enter()` объекта монитора, передавая ему объект, который нужно заблокировать:

```
Monitor.Enter(this);
```

Если монитор недоступен, значит, объект, защищенный монитором, в данный момент используется. Программа может пока заняться другой работой и через некоторое время повторить попытку. Ожидание может быть реализовано явно с помощью метода Wait(), приостанавливающего поток до тех пор, пока монитор не освободится. Метод Wait() позволяет управлять порядком выполнения потоков.

Предположим в качестве примера, что нужно загрузить из Сети статью и распечатать ее. Из соображений эффективности поместим печать в фоновый поток, но он должен быть запущен не ранее, чем будут загружены первые 10 страниц.

Печатающий поток будет ждать, пока поток, загружающий файл, не подаст сигнал, что первая порция файла загружена. Объединять эти потоки нежелательно, поскольку загружаемый файл может оказаться очень большим и ждать окончания загрузки придется очень долго. В то же время подождать загрузки первых 10 страниц все-таки необходимо. Комплекс этих требований можно удовлетворить с помощью метода Wait().

Для имитации загрузки и печати перепишем демонстрационную программу и вернем на место метод Decrementer(). Метод Incrementer() будет вести отсчет до 10, а Decrementer() — обратный отсчет до 0. Не нужно начинать обратный отсчет, пока значение переменной counter не достигнет хотя бы 5.

В методе Decrementer() выполняется вызов метода Wait() объекта Monitor:

```
if (counter < 5)
{
    Monitor.Wait(this);
}
```

Этот вызов не освобождает монитор, но сигнализирует среде CLR, что монитор понадобится программе, когда освободится. Ждущие потоки будут уведомлены о возможности попытаться продолжить работу, когда активный поток вызовет метод Pulse():

```
Monitor.Pulse(this);
```

Этот метод сообщает среде CLR, что в состоянии программы произошли изменения, способные освободить очередной ждущий поток. Среда CLR организует очередь потоков в порядке поступления запросов от них. («Ваш запрос важен для нас, и он будет обработан в порядке очереди».)

Когда поток освободит монитор, он может обозначить конец управляемой области, вызвав метод Exit():

```
Monitor.Exit(this);
```

В примере 20.4 программа-имитатор изменяется, и доступ к переменной counter синхронизируется с помощью объекта Monitor.

Пример 20.4. Использование объекта Monitor

```
namespace Programming_CSharp
{
   using System;
   using System.Threading;

   class Tester
   {
      static void Main()
      {
         // создать объект данного класса
         Tester t = new Tester();

         // выполнить за пределами статического метода Main()
         t.DoTest();
      }

      public void DoTest()
      {
         // создать массив безымянных потоков
         Thread[] myThreads =
            {
               new Thread( new ThreadStart(Decrementer) ),
               new Thread( new ThreadStart(Incrementer) )
            };

         // запустить каждый поток
         int ctr = 1;
         foreach (Thread myThread in myThreads)
         {
            myThread.IsBackground=true;
            myThread.Start();
            myThread.Name = "Thread" + ctr.ToString();
            ctr++;
            Console.WriteLine("Started thread {0}", myThread.Name);
            Thread.Sleep(50);
         }

         // подождать, пока все потоки закончат свою работу
         foreach (Thread myThread in myThreads)
         {
            myThread.Join();
         }

         // когда все потоки закончатся, вывести сообщение
         Console.WriteLine("All my threads are done.");
      }

      void Decrementer()
      {
         try
         {
            // синхронизировать этот фрагмент программы
```

```
            Monitor.Enter(this);

            // если счетчик еще не дошел до 10,
            // освободить монитор для других ждущих потоков,
            // но ждать своей очереди
            if (counter < 10)
            {
               Console.WriteLine(
                  "[{0}] In Decrementer. Counter: {1}. Gotta Wait!",
                  Thread.CurrentThread.Name, counter);
               Monitor.Wait(this);
            }

            while (counter >0)
            {
               long temp = counter;
               temp--;
               Thread.Sleep(1);
               counter = temp;
               Console.WriteLine(
                  "[{0}] In Decrementer. Counter: {1}. ",
                  Thread.CurrentThread.Name, counter);

            }
         }
         finally
         {
            Monitor.Exit(this);
         }
      }

      void Incrementer()
      {
         try
         {
            Monitor.Enter(this);
            while (counter < 10)
            {
               long temp = counter;
               temp++;
               Thread.Sleep(1);
               counter = temp;
               Console.WriteLine(
                  "[{0}] In Incrementer. Counter: {1}",
                  Thread.CurrentThread.Name, counter);
            }

            // инкрементирование выполнено, пусть теперь другой
            // поток получит монитор
            Monitor.Pulse(this);
         }
         finally
         {
```

```
            Console.WriteLine("[{0}] Exiting...",
               Thread.CurrentThread.Name);
            Monitor.Exit(this);
         }
      }
      private long counter = 0;
   }
}
```

Вывод:

```
Started thread Thread1
[Thread1] In Decrementer. Counter: 0. Gotta Wait!
Started thread Thread2
[Thread2] In Incrementer. Counter: 1
[Thread2] In Incrementer. Counter: 2
[Thread2] In Incrementer. Counter: 3
[Thread2] In Incrementer. Counter: 4
[Thread2] In Incrementer. Counter: 5
[Thread2] In Incrementer. Counter: 6
[Thread2] In Incrementer. Counter: 7
[Thread2] In Incrementer. Counter: 8
[Thread2] In Incrementer. Counter: 9
[Thread2] In Incrementer. Counter: 10
[Thread2] Exiting...
[Thread1] In Decrementer. Counter: 9.
[Thread1] In Decrementer. Counter: 8.
[Thread1] In Decrementer. Counter: 7.
[Thread1] In Decrementer. Counter: 6.
[Thread1] In Decrementer. Counter: 5.
[Thread1] In Decrementer. Counter: 4.
[Thread1] In Decrementer. Counter: 3.
[Thread1] In Decrementer. Counter: 2.
[Thread1] In Decrementer. Counter: 1.
[Thread1] In Decrementer. Counter: 0.
All my threads are done.
```

В этом примере первым стартует метод `Decrementer()`. Из выведенного текста видно, как запускается поток `Thread1` (с методом `Decrementer()`), который тут же приостанавливается. Затем запускается поток `Thread2`. Проработав до конца, он позволяет потоку `Thread1` начать работу.

Попробуем поэкспериментировать с этой программой. Во-первых, закомментируем вызов метода `Pulse()`. Окажется, что поток `Thread1` так и не возобновляет свою работу. Нет метода `Pulse()` — нет сигнала ждущим потокам.

Теперь попробуем переписать метод `Incrementer()` так, чтобы он вызывал `Pulse()` и `Exit()` после каждой операции инкрементирования:

```
void Incrementer()
{
   try
```

```
{
   while (counter < 10)
   {
    Monitor.Enter(this);
     long temp = counter;
      temp++;
      Thread.Sleep(1);
      counter = temp;
      Console.WriteLine(
         "[{0}] In Incrementer. Counter: {1}",
         Thread.CurrentThread.Name, counter);
      Monitor.Pulse(this);
      Monitor.Exit(this);
   }
```

Перепишем и метод `Decrementer()`, заменив оператор `if` на цикл `while` и уменьшив проверяемое значение счетчика с 10 до 5:

```
//if (counter < 10)
while (counter < 5)
```

В результате этих изменений поток `Thread2` (с методом `Incrementer()`) будет отпускать поток `Thread1` после каждой операции инкрементирования. Пока значение счетчика меньше пяти, метод `Decrementer()` вынужден ждать; как только оно превысит это число, метод `Decrementer()` приступит к работе. Доведя счетчик до нуля, он позволит методу `Incrementer()` продолжить выполнение. Выводимый программой текст выглядит так:

```
[Thread2] In Incrementer. Counter: 2
[Thread1] In Decrementer. Counter: 2. Gotta Wait!
[Thread2] In Incrementer. Counter: 3
[Thread1] In Decrementer. Counter: 3. Gotta Wait!
[Thread2] In Incrementer. Counter: 4
[Thread1] In Decrementer. Counter: 4. Gotta Wait!
[Thread2] In Incrementer. Counter: 5
[Thread1] In Decrementer. Counter: 4.
[Thread1] In Decrementer. Counter: 3.
[Thread1] In Decrementer. Counter: 2.
[Thread1] In Decrementer. Counter: 1.
[Thread1] In Decrementer. Counter: 0.
[Thread2] In Incrementer. Counter: 1
[Thread2] In Incrementer. Counter: 2
[Thread2] In Incrementer. Counter: 3
[Thread2] In Incrementer. Counter: 4
[Thread2] In Incrementer. Counter: 5
[Thread2] In Incrementer. Counter: 6
[Thread2] In Incrementer. Counter: 7
[Thread2] In Incrementer. Counter: 8
[Thread2] In Incrementer. Counter: 9
[Thread2] In Incrementer. Counter: 10
```

Состояние гонки и взаимные блокировки

Библиотека .NET предоставляет такую значительную поддержку потокам, какую было бы трудно обеспечить самостоятельно, создавая потоки и управляя синхронизацией вручную.

Синхронизация потоков является далеко не тривиальной задачей, особенно в сложных программах. Если читатель все же решит создавать потоки «своими руками», он столкнется с типичными проблемами, такими как состояние гонки и взаимная блокировка.

Состояние гонки

Состояние гонки (*race conditions*) создается, когда работа программы зависит от неуправляемого порядка выполнения двух независимых потоков.

Предположим, что имеются два потока, один из которых отвечает за открытие файла, а другой – за запись в него. Программист должен управлять вторым потоком так, чтобы второй поток был уверен, что первый открыл файл. Если такое управление отсутствует, то возможны два варианта. В некоторых случаях первый поток открывает файл, и второй работает нормально. При других, непредсказуемых условиях второй поток попытается писать в файл еще до того, как первый откроет его. Будет вызвано исключение, или, хуже того, программа просто закончится аварийно. Так создается состояние гонки, существенно затрудняющее отладку программы.

Нельзя допустить, чтобы эти два потока работали независимо друг от друга. Программист должен гарантировать, что поток `Thread2` не начнет свою работу до тех пор, пока не завершится выполнение потока `Thread1`. С этой целью можно объединить их методом `Join()`. В качестве альтернативы можно воспользоваться объектом `Monitor` и его методом `Wait()`, чтобы поток `Thread2` подождал подходящих условий.

Взаимная блокировка

Ожидая освобождения ресурса, поток рискует попасть в ситуацию *взаимной блокировки* (*deadlock*), которую также называют *тупиковой ситуацией* и *смертельными объятиями* (*deadly embrace*). При взаимной блокировке два или более потоков ждут друг друга, и ни один не может освободиться.

Предположим, имеются два потока, `ThreadA` и `ThreadB`. Поток `ThreadA` блокирует объект `Employee` и пытается заблокировать запись в базе данных. Выясняется, что запись заблокирована потоком `ThreadB`, поэтому поток `ThreadA` приостанавливается и ждет.

К сожалению, поток `ThreadB` не может обновить запись в базе данных, пока не обратится к объекту `Employee`, заблокированному потоком `Thre-`

adA. Ни один из двух потоков не может продолжать работу, и ни один не разблокирует захваченный ресурс. Они ждут друг друга в ситуации *взаимной блокировки*.

В этом примере распознать и устранить взаимную блокировку нетрудно. Если же в программе одновременно работает несколько потоков, диагностика, а тем более устранение взаимной блокировки представляет собой серьезную проблему. Одной из возможных рекомендаций является следующая: предоставить все требуемые блокировки или снять все имеющиеся. Иными словами, когда поток ThreadA выясняет, что не может заблокировать запись в базе данных, он должен снять блокировку с объекта Employee. Аналогичным образом, обнаружив, что объект Employee заблокирован, поток ThreadB должен освободить запись. Вторая важная рекомендация сводится к тому, что нужно блокировать как можно меньшие фрагменты программы и на как можно более короткий срок.

21

Потоки данных

Многие приложения хранят данные в памяти и обращаются с ними как с неделимыми элементами информации. Когда приложению необходима переменная или объект, оно называет нужное имя и получает то, что хочет. Однако при записи данных в файл или передаче их по локальной сети или в Интернете они должны быть организованы в *потоки данных* (*streams*). Внутри потока данные следуют друг за другом, как пузырьки воздуха в потоке воды.

Конечной точкой потока является какое-то запоминающее устройство. Оно служит источником данных, как озеро, из которого вытекает река. В типичном случае в качестве такого устройства выступает файл, но это может быть и локальная сеть, и веб-соединение.

В .NET Framework файлы и каталоги абстрагируются классами, которые предоставляют программисту методы и свойства для создания, именования, обработки и удаления файлов и каталогов на диске.

Платформа .NET предоставляет поддержку как буферизованным, так и небуферизованным потокам, а также предоставляет классы для асинхронного ввода/вывода. Асинхронный ввод/вывод позволяет программе выдать классам задание на чтение файла и, пока они его выполняют, заняться другой работой. Когда задание выполнено, классы асинхронного ввода/вывода сообщают об этом программе. Эти классы настолько мощны и устойчивы, что избавляют программиста от явного создания вычислительных потоков (см. главу 20).

Поток данных, направленный в файл или из файла, ничем не отличается от потока данных, идущего по сети, поэтому во второй части этой главы описываются манипуляции с потоками с применением как веб-протоколов, так и TCP/IP.

Чтобы создать поток данных, объект необходимо *сериализовать*, то есть вывести в поток в виде последовательности битов. Тема сериализации уже была затронута в главе 19. Платформа .NET Framework предоставляет полную поддержку сериализации, и заключительная часть главы посвящена вопросам управления сериализацией объекта.

Файлы и каталоги

Прежде чем приступить к обсуждению ввода/вывода данных в файл, рассмотрим средства, предоставляемые платформой .NET для работы с файлами и каталогами.

Классы, необходимые для этих целей, находятся в пространстве имен `System.IO`. В их число входят класс `File`, представляющий файл на диске, и класс `Directory`, который представляет каталог (или, в терминах Windows, *папку*).

Работа с каталогами

Класс `Directory` содержит статические методы, позволяющие создавать, перемещать и исследовать каталоги. Поскольку все методы этого класса являются статическими, их можно вызывать, не создавая объект класса.

Класс `DirectoryInfo` аналогичен предыдущему, но он обладает только нестатическими элементами (то есть статических элементов у него нет). Этот класс является потомком класса `FileSystemInfo`, который, в свою очередь, является потомком класса `MarshalByRefObject`. Класс `FileSystemInfo` имеет ряд свойств и методов, предоставляющих информацию о файле или каталоге.

В табл. 21.1 перечислены основные методы класса `Directory`, а табл. 21.2 содержит методы и свойства класса `DirectoryInfo`, в том числе унаследованные от класса `FileSystemInfo`.

Таблица 21.1. Основные методы класса Directory

Метод	Описание
`CreateDirectory()`	Создает все каталоги и подкаталоги пути, указанного в аргументе
`Delete()`	Удаляет каталог со всем его содержимым
`Exists()`	Возвращает логическое значение, равное `true`, если путь, указанный в аргументе, ведет к существующему каталогу
`GetCreationTime()`	Возвращает дату и время создания каталога
`SetCreationTime()`	Устанавливает дату и время создания каталога
`GetCurrentDirectory()`	Возвращает текущий каталог

Метод	Описание
SetCurrentDirectory()	Устанавливает текущий каталог
GetDirectories()	Возвращает массив подкаталогов
GetDirectoryRoot()	Возвращает корень для пути, указанного в аргументе
GetFiles()	Возвращает массив строк с именами файлов в указанном каталоге
GetLastAccessTime()	Возвращает время последнего обращения к указанному каталогу
SetLastAccessTime()	Устанавливает время последнего обращения к указанному каталогу
GetLastWriteTime()	Возвращает время последней записи в указанный каталог
SetLastWriteTime()	Устанавливает время последней записи в указанный каталог
GetLogicalDrives()	Возвращает имена всех логических дисков в формате <drivel>:\
GetParent()	Возвращает родительский каталог для указанного пути
Move()	Перемещает каталог со всем содержимым в соответствии с указанным путем

Таблица 21.2. Основные методы и свойства класса DirectoryInfo

Метод или свойство	Описание
Attributes	Наследуется от FileSystemInfo; возвращает или устанавливает атрибуты текущего файла
CreationTime	Наследуется от FileSystemInfo; возвращает или устанавливает время создания текущего файла
Exists	Открытое свойство логического типа; равно true, если каталог существует
Extension	Открытое свойство, наследуемое от FileSystemInfo; представляет собой расширение файла
FullName	Открытое свойство, наследуемое от FileSystemInfo; представляет собой полный путь к файлу или каталогу
LastAccessTime	Открытое свойство, наследуемое от FileSystemInfo; возвращает или устанавливает время последнего доступа
LastWriteTime	Открытое свойство, наследуемое от FileSystemInfo; возвращает или устанавливает время последней записи в текущий файл или каталог
Name	Открытое свойство; имя данного экземпляра Directory Info
Parent	Открытое свойство; родительский каталог указанного подкаталога

Таблица 21.2 (продолжение)

Метод или свойство	Описание
Root	Открытое свойство; корень из указанного пути
Create()	Открытый метод, создающий каталог
CreateSubdirectory()	Открытый метод, создающий подкаталог по заданному пути
Delete()	Открытый метод, удаляющий объект DirectoryInfo и его содержимое из указанного пути
GetDirectories()	Открытый метод, возвращающий массив элементов DirectoryInfo с подкаталогами
GetFiles()	Открытый метод, возвращающий список файлов в каталоге
GetFileSystemInfos()	Открытый метод, возвращающий массив объектов типа FileSystemInfo
MoveTo()	Открытый метод, перемещающий объект DirectoryInfo и все его содержимое в соответствии с указанным путем
Refresh()	Открытый метод, наследуемый от FileSystemInfo; обновляет состояние объекта

Создание объекта DirectoryInfo

Чтобы исследовать иерархическую структуру каталогов, необходимо создать объект класса DirectoryInfo. Этот класс предоставляет программисту методы, позволяющие получить не только имена файлов и подкаталогов, но и объекты FileInfo и DirectoryInfo. С помощью последних программист исследует иерархическую структуру каталога, рекурсивно извлекая его подкаталоги.

При создании объекта класса DirectoryInfo следует указать имя рассматриваемого каталога:

```
string path=Enviroment.GetEnviromentVariable("SystemRoot");
DirectoryInfo dir = new DirectoryInfo(path);
```

Читатель помнит, что символ @ перед строкой создает дословный строковый литерал, в котором не нужно использовать управляющие последовательности, такие как обратный слэш (см. главу 10).

У объекта DirectoryInfo можно запросить информацию о нем самом, в том числе его имя, полный путь, атрибуты, время последнего обращения и т. д. Чтобы исследовать иерархическую структуру текущего каталога, следует запросить у него список подкаталогов:

```
DirectoryInfo[] directories = dir.GetDirectories();
```

Метод GetDirectories() возвращает массив объектов типа DirectoryInfo, каждый из которых представляет собой каталог. Этот метод можно вызывать рекурсивно, поочередно передавая ему элементы возвращенного массива:

```
foreach (DirectoryInfo newDir in directories)
{
   dirCounter++;
   ExploreDirectory(newDir);
}
```

Статическая целочисленная переменная dirCounter отслеживает общее число обнаруженных подкаталогов. Чтобы сделать вывод интереснее, добавим еще одну статическую целочисленную переменную indentLevel, которая будет увеличиваться на единицу при каждом рекурсивном входе в подкаталог и уменьшаться на единицу при выходе. В результате подкаталоги будут выводиться с отступом по отношению к их родительским каталогам. Полный текст программы приведен в примере 21.1.

Пример 21.1. Рекурсивный обход подкаталогов

```
namespace Programming_CSharp
{
   using System;
   using System.IO;

   class Tester
   {
      public static void Main(  )
      {
         Tester t = new Tester(  );

         // выбрать начальный подкаталог
         string theDirectory =
            Environment.GetEnvironmentVariable("SystemRoot");

         // вызвать метод, исследующий структуру каталога
         // и выводящий дату обращения к нему, а также
         // все его подкаталоги
         DirectoryInfo dir = new DirectoryInfo(theDirectory);

         t.ExploreDirectory(dir);

         // работа закончена; вывести статистику

         Console.WriteLine(
            "\n\n{0} directories found.\n",
            dirCounter);
      }

      // при запуске передать методу объект DirectoryInfo;
      // для каждого найденного каталога он вызовет себя рекурсивно
      private void ExploreDirectory(DirectoryInfo dir)
      {
         indentLevel++;  // понизить уровень для каталогов
```

```
            // создать отступы для подкаталогов
            for (int i = 0; i < indentLevel; i++)
               Console.Write("  "); // два пробела на уровень

            // вывести каталог и время последнего обращения
            Console.WriteLine("[{0}] {1} [{2}]\n",
               indentLevel, dir.Name, dir.LastAccessTime);

            // получить все подкаталоги текущего каталога
            // и рекурсивно вызвать метод для каждого из них
            DirectoryInfo[] directories = dir.GetDirectories(  );
            foreach (DirectoryInfo newDir in directories)
            {
               dirCounter++;  // увеличить счетчик
               ExploreDirectory(newDir);
            }
            indentLevel--; // повысить уровень для каталогов
         }

         // статические переменные класса, отслеживающие
         // количество подкаталогов и величину отступа
         static int dirCounter = 1;
         static int indentLevel  = -1; // so first push = 0
      }
   }
```

Вывод (отрывок):

```
    [2] logiscan [5/1/2001 3:06:41 PM]

    [2] miitwain [5/1/2001 3:06:41 PM]

  [1] Web [5/1/2001 3:06:41 PM]

    [2] printers [5/1/2001 3:06:41 PM]

      [3] images [5/1/2001 3:06:41 PM]

    [2] Wallpaper [5/1/2001 3:06:41 PM]

363 directories found.
```

Программа начинается с идентификации каталога *%SystemRoot%*, (обычно *C:\WinNT* или *C:\Windows*) и создания для него объекта `DirectoryInfo`. Затем следует вызов метода `ExploreDirectory()`, которому в качестве аргумента передается только что созданный объект. Метод выводит информацию о каталоге и просматривает его подкаталоги.

Список всех подкаталогов текущего каталога можно получить, вызвав метод `GetDirectories()`. Он возвращает массив объектов типа `DirectoryInfo`. Метод `ExploreDirectory()` является рекурсивным; он вызывает себя для каждого объекта из массива. В результате он то погружается в каждый подкаталог, то выходит из него для обследования подкаталогов того же уровня. Это продолжается, пока не будут выведены все подкаталоги каталога *%SystemRoot%*. Когда метод `ExploreDirectory()` в

конце концов возвращает управление, вызвавший метод выводит статистику.

Работа с файлами

Объект DirectoryInfo может, кроме всего прочего, возвратить коллекцию всех файлов в каждом найденном подкаталоге. Метод GetFiles() возвращает массив объектов FileInfo, каждый из которых описывает файл. Объекты FileInfo и File связаны друг с другом аналогично тому, как связаны DirectoryInfo и Directory. Как и методы класса Directory, все методы класса File – статические; подобно методам класса DirectoryInfo, все методы класса FileInfo являются нестатическими.

В табл. 21.3 перечислены основные методы класса File, а в табл. 21.4 – самые важные элементы класса FileInfo.

Таблица 21.3. Основные открытые статические методы класса File

Метод	Описание
AppendText()	Создает объект StreamWriter, который дописывает текст в конец указанного файла
Copy()	Копирует существующий файл в новый
Create()	Создает файл в соответствии с указанным путем
CreateText()	Создает объект StreamWriter, который записывает новый текст в указанный файл
Delete()	Удаляет указанный файл
Exists()	Возвращает true, если указанный файл существует
GetAttributes()	Возвращает атрибуты указанного файла
SetAttributes()	Устанавливает атрибуты указанного файла
GetCreationTime()	Возвращает дату и время создания файла
SetCreationtime()	Устанавливает дату и время создания файла
GetLastAccessTime()	Возвращает время последнего обращения к указанному файлу
SetLastAccessTime()	Устанавливает время последнего обращения к указанному файлу
GetLastWriteTime()	Возвращает время последней записи в указанный файл
SetLastWriteTime()	Устанавливает время последней записи в указанный файл
Move()	Переносит файл в новое место; может быть использован для переименования файла
OpenRead()	Открытый статический метод, который открывает поток для файла
OpenWrite()	Создает поток чтения/записи в соответствии с указанным путем

Таблица 21.4. Методы и свойства класса FileInfo

Метод или свойство	Описание
Attributes	Наследуется от FileSystemInfo; возвращает или устанавливает атрибуты текущего файла
CreationTime	Наследуется от FileSystemInfo; возвращает или устанавливает время создания текущего файла
Directory	Открытое свойство, возвращающее экземпляр родительского каталога
Exists	Открытое свойство логического типа; равно true, если каталог существует
Extension	Открытое свойство, наследуемое от FileSystemInfo; представляет собой расширение файла
FullName	Открытое свойство, наследуемое от FileSystemInfo; представляет собой полный путь к файлу или каталогу
LastAccessTime	Открытое свойство, наследуемое от FileSystemInfo; возвращает или устанавливает время последнего доступа
LastWriteTime	Открытое свойство, наследуемое от FileSystemInfo; возвращает или устанавливает время последней записи в текущий файл или каталог
Length	Открытое свойство, возвращающее размер текущего файла
Name	Открытое свойство; имя данного экземпляра FileInfo
AppendText()	Открытый метод, создающий объект StreamWriter, который дописывает текст в конец файла
CopyTo()	Открытый метод, который копирует существующий файл в новый файл
Create()	Открытый метод, создающий новый файл
Delete()	Открытый метод, удаляющий файл
MoveTo()	Открытый метод, который переносит файл в новое место; может быть использован для переименования файла
Open()	Открытый метод, который открывает файл с различными привилегиями на чтение/запись и совместное использование
OpenRead()	Открытый метод, создающий поток только для чтения
OpenText()	Открытый метод, создающий объект StreamReader, который читает из существующего текстового файла
OpenWrite()	Открытый метод, создающий поток только для записи

Пример 21.2 представляет собой измененную версию примера 21.1. Добавлен программный текст, получающий объект FileInfo для каждого файла в каталоге. Эти объекты предоставляют информацию, на основании которой программа выводит имя каждого файла, его длину и дату/время последнего обращения.

Пример 21.2. Исследование файлов и поддиректориев

```
namespace Programming_CSharp
{
   using System;
   using System.IO;

   class Tester
   {
      public static void Main(  )
      {
         Tester t = new Tester(  );

         // выбрать начальный подкаталог
         string theDirectory =
            Environment.GetEnvironmentVariable("SystemRoot");

         // вызвать метод, исследующий структуру каталога
         // и выводящий дату обращения к нему, а также
         // все его подкаталоги
         DirectoryInfo dir = new DirectoryInfo(theDirectory);

         t.ExploreDirectory(dir);

         // закончено; вывести статистику
         Console.WriteLine(
            "\n\n{0} files in {1}  directories found.\n",
            fileCounter,dirCounter);
      }

      // при запуске передать методу объект directoryInfo;
      // для каждого найденного каталога он вызовет себя
      // рекурсивно
      private void ExploreDirectory(DirectoryInfo dir)
      {
         indentLevel++;  // понизить уровень каталога
         // создать отступ для подкаталога
         for (int i = 0; i < indentLevel; i++)
            Console.Write("  "); // два пробела на уровень

         // вывести каталог и время последнего доступа
         Console.WriteLine("[{0}] {1} [{2}]\n",
            indentLevel, dir.Name, dir.LastAccessTime);

         // получить все файлы из каталога, вывести
         // их имена, время последнего доступа и размеры
         FileInfo[] filesInDir = dir.GetFiles(  );
         foreach (FileInfo file in filesInDir)
         {
            // один дополнительный отступ для вывода файлов
            for (int i = 0; i < indentLevel+1; i++)
               Console.Write("  "); // два пробела на уровень

            Console.WriteLine("{0} [{1}] Size: {2} bytes",
               file.Name,
```

```
                file.LastWriteTime,
                file.Length);
            fileCounter++;
        }

        // получить все подкаталоги текущего каталога и
        // рекурсивно вызвать метод для каждого из них
        DirectoryInfo[] directories = dir.GetDirectories(  );
        foreach (DirectoryInfo newDir in directories)
        {
            dirCounter++;  // увеличить счетчик
            ExploreDirectory(newDir);
        }
        indentLevel--; // повысить уровень каталога
    }

    // статические переменные класса, отслеживающие
    // количество подкаталогов и величину отступа
    static int dirCounter = 1;
    static int indentLevel  = -1; // so first push = 0
    static int fileCounter = 0;
  }
}
```

Вывод (отрывок):

```
[0] WinNT [5/1/2001 3:34:01 PM]

  Active Setup Log.txt [4/20/2001 10:42:22 AM] Size: 10620 bytes
  actsetup.log [4/20/2001 12:05:02 PM] Size: 8717 bytes
  Blue Lace 16.bmp [12/6/1999 4:00:00 PM] Size: 1272 bytes
    [2] Wallpaper [5/1/2001 3:14:32 PM]
      Boiling Point.jpg [4/20/2001 8:30:24 AM] Size: 28871 bytes
      Chateau.jpg [4/20/2001 8:30:24 AM] Size: 70605 bytes
      Windows 2000.jpg [4/20/2001 8:30:24 AM] Size: 129831 bytes

8590 files in 363  directories found.
```

Эта программа получает имя каталога *%SystemRoot%*. Она выводит информацию обо всех файлах в этом каталоге и затем с помощью рекурсивных вызовов исследует все его подкаталоги (на компьютере читателя выведенный текст будет отличаться). Выполнение программы займет немало времени, поскольку дерево подкаталогов каталога *%SystemRoot%* достаточно велико (на компьютере автора 363 подкаталога).

Изменение файлов

Как видно из табл. 21.3 и 21.4, с помощью класса `FileInfo` можно создавать, копировать, переименовывать и удалять файлы. В следующем примере создается подкаталог, в который будут скопированы файлы. Некоторые из них будут переименованы, другие – удалены. В конце программы будет удален и сам подкаталог.

Прежде чем выполнить следующие примеры, создайте каталог *test* и скопируйте в него каталог *media* из каталога *WinNT* или *Windows*. Не работайте непосредственно в системном каталоге, так как обращение с системными файлами требует исключительной осторожности.

Первый шаг состоит в создании объекта `DirectoryInfo` для тестового каталога:

```
string theDirectory = @"c:\test\media";
DirectoryInfo dir = new DirectoryInfo(theDirectory);
```

Теперь создадим подкаталог внутри тестового каталога, вызвав метод `CreateSubDirectory()` объекта `DirectoryInfo`. Метод возвратит новый объект типа `DirectoryInfo`, который представляет только что созданный каталог:

```
string newDirectory = "newTest";
DirectoryInfo newSubDir = dir.CreateSubdirectory(newDirectory);
```

Теперь можно просмотреть в цикле содержимое тестового каталога и скопировать файлы в созданный нами подкаталог:

```
FileInfo[] filesInDir = dir.GetFiles();
foreach (FileInfo file in filesInDir)
{
   string fullName = newSubDir.FullName +
      "\\" + file.Name;
   file.CopyTo(fullName);
   Console.WriteLine("{0} copied to newTest",
      file.FullName);
}
```

Обратите внимание на синтаксис метода `CopyTo()`. Это метод объекта `FileInfo`. Ему следует передать полный путь к новому файлу, в том числе его имя и расширение.

Когда копирование закончится, можно будет получить список файлов в подкаталоге и работать с ними непосредственно:

```
filesInDir = newSubDir.GetFiles();
foreach (FileInfo file in filesInDir)
{
```

Создадим простую целочисленную переменную `counter` и с ее помощью переименуем файлы, через один:

```
if (counter++ %2 == 0)
{
    file.MoveTo(fullName + ".bak");
    Console.WriteLine("{0} renamed to {1}",
        fullName,file.FullName);
}
```

Здесь файл переименовывается за счет копирования с удалением в тот же самый каталог, но с другим именем. Конечно, файл может быть перемещен в другой каталог без изменения его имени, а может быть перемещен и переименован одновременно.

Итак, файлы переименовываются через один; остальные удаляются:

```
file.Delete();
Console.WriteLine("{0} deleted.",
    fullName);
```

Закончив работу с файлами, можно «прибрать за собой», удалив и весь подкаталог:

```
newSubDir.Delete(true);
```

Логический аргумент определяет, является ли удаление рекурсивным. Если передать `false` и в данном каталоге есть подкаталоги с файлами, будет вызвано исключение.

В примере 21.3 содержится текст обсуждаемой программы. Выполняя его, будьте осторожны: по окончании программы каталог удаляется. Чтобы проследить за ходом переименований и удалений, либо поставьте точку останова на последней строчке программы, либо вообще удалите эту строчку.

Пример 21.3. Создание подкаталога и работа с файлами

```
namespace Programming_CSharp
{
   using System;
   using System.IO;

   class Tester
   {
      public static void Main(  )
      {
         // создать объект класса и выполнить его
         Tester t = new Tester(  );
         string theDirectory = @"c:\test\media";
         DirectoryInfo dir = new DirectoryInfo(theDirectory);
         t.ExploreDirectory(dir);
      }

      // запустить, передав имя каталога
      private void ExploreDirectory(DirectoryInfo dir)
      {

         // создать новый подкаталог
         string newDirectory = "newTest";
         DirectoryInfo newSubDir = dir.CreateSubdirectory(newDirectory);

         // получить все файлы каталога и
         // скопировать их в новый каталог
```

```
            FileInfo[] filesInDir = dir.GetFiles(  );
            foreach (FileInfo file in filesInDir)
            {
               string fullName = newSubDir.FullName +
                  "\\" + file.Name;
               file.CopyTo(fullName);
               Console.WriteLine("{0} copied to newTest", file.FullName);
            }

            // получить коллекцию скопированных файлов
            filesInDir = newSubDir.GetFiles(  );

            // одни файлы переименовать, другие - удалить
            int counter = 0;
            foreach (FileInfo file in filesInDir)
            {
               string fullName = file.FullName;

               if (counter++ %2 == 0)
               {
                  file.MoveTo(fullName + ".bak");
                  Console.WriteLine("{0} renamed to {1}",
                     fullName,file.FullName);
               }
               else
               {
                  file.Delete(  );
                  Console.WriteLine("{0} deleted.",
                     fullName);
               }
            }

            newSubDir.Delete(true); // delete the subdirectory
         }
      }
   }
```

Вывод (отрывок):

```
c:\test\media\Bach's Brandenburg Concerto No. 3.RMI
        copied to newTest
c:\test\media\Beethoven's 5th Symphony.RMI copied to newTest
c:\test\media\Beethoven's Fur Elise.RMI copied to newTest
c:\test\media\canyon.mid copied to newTest
c:\test\media\newTest\Bach's Brandenburg Concerto
        No. 3.RMI renamed to
c:\test\media\newTest\Bach's Brandenburg Concerto
        No. 3.RMI.bak
c:\test\media\newTest\Beethoven's 5th Symphony.RMI deleted.
c:\test\media\newTest\Beethoven's Fur Elise.RMI renamed to
c:\test\media\newTest\Beethoven's Fur Elise.RMI.bak
c:\test\media\newTest\canyon.mid deleted.
```

Чтение и запись данных

Чтение и запись данных выполняются при помощи класса Stream. В начале главы речь уже шла о потоках данных. По большому счету, им посвящена вся эта глава.[1] Класс Stream поддерживает синхронный и асинхронный ввод/вывод данных. Платформа .NET Framework предоставляет программистам целый ряд классов, являющихся потомками Stream, в том числе FileStream, MemoryStream и NetworkStream. Кроме них существует еще класс BufferedStream, который позволяет реализовать буферизованный ввод/вывод и может быть использован с любым другим классом потока. Основные классы, применяемые при вводе/выводе, собраны в табл. 21.5.

Таблица 21.5. Основные классы ввода/вывода, предоставляемые платформой .NET Framework

Класс	Описание
Stream	Абстрактный класс, поддерживающий чтение и запись байтов
BinaryReader/ BinaryWriter	Чтение и запись в поток закодированных строк и базовых типов данных
File, FileInfo, Directory, DirectoryInfo	Предоставляют реализацию абстрактных классов FileSystemInfo, в том числе создание, перемещение, переименование и удаление файлов и каталогов
FileStream	Предназначен для чтения/записи объектов класса File; поддерживает произвольный доступ к файлам. По умолчанию открывает файлы синхронно, поддерживает асинхронный доступ к файлам
TextReader, TextWriter, StringReader, StringWriter	TextReader и TextWriter являются абстрактными классами, предназначенными для ввода/вывода символов Unicode. Классы StringReader и StringWriter выполняют чтение/запись в строки, что позволяет при вводе/выводе пользоваться либо потоком данных, либо строкой
BufferedStream	Поток данных, «добавляющий» буферизацию к другому потоку, например NetworkStream. Обратите внимание, что класс File Stream имеет встроенную буферизацию. Классы BufferedStream могут повысить производительность потоков, к которым они прикреплены
MemoryStream	Небуферизованный поток, инкапсулированные данные которого непосредственно доступны в памяти. Класс MemoryStream не имеет внешней памяти и наиболее полезен в качестве временного буфера
NetworkStream	Поток данных в сетевом соединении

[1] Автор снимает шляпу перед Арло Гутри (Arlo Guthrie).

Двоичные файлы

В начале этого раздела базовый класс Stream будет использован для чтения двоичного файла. Термин *двоичный файл* применяется как антоним термина *текстовый файл*. Если о файле не известно наверняка, что он является текстовым, безопаснее считать его потоком байтов, который и называется *двоичным файлом*.

Класс Stream «до отказа» заполнен методами, но самыми важными являются Read(), Write(), BeginRead(), BeginWrite() и Flush(). Все они подробно рассматриваются в следующих разделах.

Чтобы выполнить чтение двоичного файла, создадим два объекта класса Stream, один для чтения, а другой для записи.

```
Stream inputStream = File.OpenRead(@"C:\test\source\test1.cs");

Stream outputStream = File.OpenWrite(@"C:\test\source\test1.bak");
```

С помощью статических методов OpenRead() и OpenWrite() класса File файлы открываются соответственно для чтения и записи. Статические версии этих перегруженных методов принимают в качестве аргумента путь к файлу.

Чтение двоичного файла всегда выполняется через буфер. Буфер – это просто байтовый массив, содержащий данные, прочитанные методом Read().

Программист передает методу буфер, смещение в буфере, с которого следует сохранять считываемые данные, и число считываемых байтов. Метод InputStream.Read() читает байты из внешней памяти, сохраняет их в буфере и возвращает количество считанных байт.

Он продолжает читать, пока не прочитает все данные:

```
while ( (bytesRead = inputStream.Read(buffer,0,SIZE_BUFF)) > 0 )
{
    outputStream.Write(buffer,0,bytesRead);
}
```

Заполненный буфер записывается в выходной файл. Аргументами метода Write() являются буфер, из которого он берет информацию, смещение в этом буфере, с которого следует выводить данные, и число записываемых байт. Нетрудно заметить, что в программе будет выведено столько же байт, сколько прочитано.

Полный текст программы приведен в примере 21.4.

Пример 21.4. Реализация ввода/вывода в двоичный файл

```
namespace Programming_CSharp
{
   using System;
   using System.IO;
```

```
    class Tester
    {
       const int SizeBuff = 1024;

       public static void Main()
       {
          // создать объект класса и выполнить его
          Tester t = new Tester();
          t.Run();
       }

       // запустить с именем директория
       private void Run()
       {
          // файл для чтения
          Stream inputStream = File.OpenRead(
             @"C:\test\source\test1.cs");

          // файл для записи
          Stream outputStream = File.OpenWrite(
             @"C:\test\source\test1.bak");

          // создать буфер для хранения байтов
          byte[] buffer = new Byte[SizeBuff];
          int bytesRead;

          // пока метод Read() возвращает байты,
          // записывать их в выходной поток
          while ( (bytesRead = inputStream.Read(buffer,0,SizeBuff)) > 0 )
          {
             outputStream.Write(buffer,0,bytesRead);
          }

          // "прибрать за собой" перед выходом из программы
          inputStream.Close();
          outputStream.Close();
       }
    }
}
```

Результатом работы этой программы будет копия входного файла (*test1.cs*), расположенная в том же каталоге и названная *test1.bak*.

Буферизация потоков

В предыдущем примере был создан буфер для чтения. При вызове метода `Read()` буфер заполняется данными, считанными с диска. Однако в некоторых случаях эффективность работы операционной системы можно повысить, если прочитать больше (или меньше) байтов за один прием.

Буферизованный поток – это объект, позволяющий операционной системе создавать собственный внутренний буфер для обмена данны-

ми с внешней памятью и самостоятельно определять число байтов, считываемых или записываемых за один раз. Она по-прежнему будет заполнять буфер программы указанными порциями, но данные в этот буфер будут поступать из внутреннего системного буфера, а не из внешней памяти. В результате ввод/вывод будет происходить эффективнее и, следовательно, быстрее.

Объект BufferedStream надстраивается над существующим объектом Stream, и, чтобы им воспользоваться, программист должен сначала создать обычный класс потока, как это делалось в примере 21.4:

```
Stream inputStream = File.OpenRead(@"C:\test\source\folder3.cs");

Stream outputStream = File.OpenWrite(@"C:\test\source\folder3.bak");
```

После этого надо передать объект класса Stream конструктору буферизованного потока:

```
BufferedStream bufferedInput = new BufferedStream(inputStream);

BufferedStream bufferedOutput = new BufferedStream(outputStream);
```

Затем с объектом класса BufferedStream можно обращаться как с обычным потоком, вызывая методы Read() и Write(), как это делалось в предыдущем примере. Операционная система сама будет выполнять буферизацию:

```
while ( (bytesRead = bufferedInput.Read(buffer,0,SIZE_BUFF)) > 0 )
 {
     bufferedOutput.Write(buffer,0,bytesRead);
 }
```

Единственной особенностью применения буферизованных потоков является необходимость принудительно освобождать буфер, чтобы гарантировать, что все данные выведены в файл:

```
bufferedOutput.Flush();
```

Метод Flush() сообщает операционной системе, что все содержимое внутреннего буфера следует вывести на диск. Пример 21.5 содержит пример использования буферизованного потока.

Пример 21.5. Ввод/вывод с буферизацией

```
namespace Programming_CSharp
{
   using System;
   using System.IO;

   class Tester
   {
      const int SizeBuff = 1024;

      public static void Main()
```

```
        {
           // создать объект класса и выполнить его
           Tester t = new Tester();
           t.Run();
        }

        // запустить с именем каталога
        private void Run()
        {
           // создать двоичные потоки
           Stream inputStream = File.OpenRead(@"C:\test\source\folder3.cs");

          Stream outputStream = File.OpenWrite(@"C:\test\source\folder3.bak");

           // надстроить буферизованные потоки над двоичными
           BufferedStream bufferedInput =
              new BufferedStream(inputStream);

           BufferedStream bufferedOutput = new BufferedStream(outputStream);
           byte[] buffer = new Byte[SizeBuff];
           int bytesRead;

           while ( (bytesRead = bufferedInput.Read(buffer,0,SizeBuff)) > 0 )
           {
              bufferedOutput.Write(buffer,0,bytesRead);
           }

           bufferedOutput.Flush();
           bufferedInput.Close();
           bufferedOutput.Close();

        }
     }
  }
```

С большими файлами эта программа будет работать быстрее, чем программа из примера 21.4.

Работа с текстовыми файлами

Если доподлинно известно, что файл содержит только текст, то для чтения или записи можно применять классы `StreamReader` и `StreamWriter` соответственно. Эти классы разработаны специально для облегчения работы с текстом. Например, они обладают методами `ReadLine()` и `WriteLine()`, позволяющими читать и записывать одну строку текста. Метод `WriteLine()` уже неоднократно использовался в этой книге с объектом `Console`.

Чтобы создать экземпляр класса `StreamReader`, необходимо сначала создать объект `FileInfo` и затем вызвать его метод `OpenText()`:

```
FileInfotheSourceFile = new FileInfo (@"C:\test\source\test1.cs");

StreamReader stream = theSourceFile.OpenText();
```

Этот метод возвращает объект StreamReader для файла. Имея этот объект, можно приступать к построчному чтению файла:

```
do
{
    text = stream.ReadLine();
} while (text != null);
```

Метод ReadLine() считывает одну строчку за каждый вызов, пока не достигнет конца файла. В этом случае объект StreamReader возвратит null.

Для того чтобы создать класс StreamWriter, надо вызвать одноименный конструктор и передать ему полное имя файла, в который следует выводить текст:

```
StreamWriter writer = new
StreamWriter(@"C:\test\source\folder3.bak",false);
```

Вторым параметром конструктора является логический аргумент append. Если файл уже существует, передача значения true приведет к тому, что новые данные будут дописываться в конец файла.

Передача значения false указывает на перезапись файла. В рассматриваемом примере передается false, чтобы файл был переписан, если он существует.

Напишем цикл, переписывающий каждую строку старого файла в новый, попутно выводя ее на экран:

```
do
{
    text = reader.ReadLine();
    writer.WriteLine(text);
    Console.WriteLine(text);
} while (text != null);
```

В примере 21.6 приведен полный текст программы.

Пример 21.6. Чтение и запись текстового файла

```
namespace Programming_CSharp
{
   using System;
   using System.IO;

   class Tester
   {
      public static void Main()
      {
         // создать объект класса и выполнить его
         Tester t = new Tester();
         t.Run();
      }

      // запустить с именем каталога
```

```
        private void Run()
        {
           // открыть файл
           FileInfo theSourceFile = new FileInfo(
              @"C:\test\source\test.cs");

           // создать класс для чтения текстового файла
           StreamReader reader = theSourceFile.OpenText();

           // создать класс для текстовой записи в новый файл
           StreamWriter writer = new StreamWriter(
              @"C:\test\source\test.bak",false);

           // создать переменную для хранения строк текста
           string text;

           // пройти по файлу и прочитать каждую строку,
           // выводя ее на экран и в файл
           do
           {
              text = reader.ReadLine();
              writer.WriteLine(text);
              Console.WriteLine(text);
           } while (text != null);

           // прибрать за собой
           reader.Close();
           writer.Close();
        }
     }
  }
```

Когда эта программа работает, содержимое исходного файла выводится как на экран, так и в новый файл. Обратите внимание на оператор вывода на экран:

```
Console.WriteLine(text);
```

Он практически совпадает с оператором записи в файл:

```
writer.WriteLine(text);
```

Главное различие между ними заключается в том, что метод `WriteLine()` объекта `Console` статический, а одноименный метод объекта `StreamWriter`, производного от `TextWriter`, является нестатическим и вызывается для объекта, а не для класса.

Асинхронный ввод/вывод

Все программы, рассмотренные до сих пор, выполняли *синхронный ввод/вывод*. Это означает, что на время ввода или вывода вся прочая деятельность программы прекращается. Однако обмен данными с внешней памятью может занять относительно много времени, если в

качестве внешней памяти выступает медленный диск или (о, ужас!) медленное сетевое соединение.

Если файл очень велик или чтение/запись выполняется в сети, разумнее обратиться к *асинхронному вводу/выводу*, который позволяет начать чтение или запись и затем переключиться на другую задачу, пока среда CLR не выполнит требуемую операцию. Платформа .NET Framework поддерживает асинхронный ввод/вывод, предоставляя методы `BeginRead()` и `BeginWrite()` класса `Stream`.

Программа вызывает метод `BeginRead()` для соответствующего файла и переходит к решению других задач, пока чтение выполняется в отдельном вычислительном потоке. Когда чтение завершится, метод обратного вызова сообщит об этом. Программа сможет обработать прочитанные данные, снова запустить чтение и заняться другой работой.

В дополнение к трем аргументам, указываемым при чтении двоичных файлов (буфер, объект, сколько байт прочитать), метод `BeginRead()` принимает *делегат* и *объект состояния*.

Делегат – это необязательный аргумент, представляющий метод обратного вызова. Если этот метод указан, он вызывается по окончании чтения данных. Объект состояния тоже является необязательным аргументом. В рассматриваемом примере методу `BeginRead()` в качестве объекта состояния передается `null`. Состояние хранится в переменных тестового класса.

Программист может передать в качестве состояния любой объект и прочитать его при обратном вызове. Как правило, в этом объекте сохраняются переменные, отслеживающие состояние (о чем нетрудно догадаться по названию формального параметра), которые понадобятся впоследствии. Параметр «состояние» может понадобиться разработчику для хранения состояния вызова (находится в паузе, приостановлен, выполняется и т. д.).

В следующем примере буфер и объект `Stream` создаются как закрытые переменные класса:

```
public class AsynchIOTester
{
    private Stream inputStream;
    private byte[] buffer;
    const int BufferSize = 256;
```

Делегат также создается как закрытый член класса:

```
private AsyncCallback myCallBack; // делегированный метод
```

Здесь объявлено, что делегат имеет тип `AsyncCallback`. Именно такой тип требуется методу `BeginRead()` класса `Stream`.

Делегат `AsyncCallBack` объявляется в пространстве имен `System`:

```
public delegate void AsyncCallback (IAsyncResult ar);
```

Таким образом, делегат может быть связан с любым методом, возвращающим void и принимающим интерфейс IAsyncResult в качестве аргумента. Среда CLR на этапе исполнения программы передаст методу объект интерфейса типа IAsyncResult. Программист должен лишь объявить этот метод:

```
void OnCompletedRead(IAsyncResult asyncResult)
```

и назначить его делегатом в конструкторе:

```
AsynchIOTester()
{
        //...
     myCallBack = new AsyncCallback(this.OnCompletedRead);
}
```

Разберем работу этого механизма. В методе Main() создается и запускается экземпляр класса:

```
public static void Main()
{
    AsynchIOTester theApp = new AsynchIOTester();
    theApp.Run();
}
```

Ключевое слово new фактически означает вызов конструктора. Конструктор открывает файл и получает объект Stream. Затем выделяется место в буфере и запускается механизм обратного вызова:

```
AsynchIOTester()
{
    inputStream = File.OpenRead(@"C:\test\source\AskTim.txt");
    buffer = new byte[BufferSize];
    myCallBack = new AsyncCallback(this.OnCompletedRead);
}
```

Для этого примера нужен большой текстовый файл. Автор скопировал статью Тима О'Рейли (Tim O'Reilly) из рубрики «Ask Tim» сайта *http://www.oreilly.com* в текстовый файл *AskTim.txt*, который поместил в подкаталог *test\source* на диске C. Читатель волен взять любой текстовый файл из любого подкаталога.

В методе Run() вызывается метод BeginRead(), выполняющий асинхронное чтение файла:

```
inputStream.BeginRead(
     buffer,              // куда помещать результат
     0,                   // смещение в буфере
     buffer.Length,       // размер буфера
     myCallBack,          // делегат обратного вызова
     null);               // локальный объект состояния
```

После этого программа может заняться другим делом. В данном примере программа имитирует работу, считая до 500 000 и выводя на экран каждое тысячное значение:

```
for (long i = 0; i < 500000; i++)
{
    if (i%1000 == 0)
    {
        Console.WriteLine("i: {0}", i);
    }
}
```

Когда чтение файла закончится, среда CLR вызовет метод обратного вызова:

```
void OnCompletedRead(IAsyncResult asyncResult)
{
```

Получив уведомление об окончании ввода, в первую очередь необходимо выяснить, сколько байт фактически прочитано. С этой целью вызывается метод `EndRead()` объекта `Stream`, которому передается объект интерфейса `IAsyncResult`, полученный от среды CLR:

```
int bytesRead = inputStream.EndRead(asyncResult);
```

Метод `EndRead()` возвращает число прочитанных байт. Если это число больше нуля, буфер преобразуется в строку и выводится на экран, после чего метод `BeginRead()` вызывается снова для выполнения следующего асинхронного ввода:

```
if (bytesRead > 0)
{
    String s =
    Encoding.ASCII.GetString (buffer, 0, bytesRead);
    Console.WriteLine(s);
    inputStream.BeginRead(
    buffer, 0, buffer.Length,
    myCallBack, null);
}
```

В результате этих действий программа получает возможность выполнять другую работу, пока происходит чтение. Что касается прочитанных данных, они обрабатываются (в этом примере – выводятся на экран) после каждого заполнения буфера. Полный текст рассмотренной программы содержится в примере 21.7.

Пример 21.7. Реализация асинхронного ввода/вывода

```
namespace Programming_CSharp
{
   using System;
   using System.IO;
   using System.Threading;
```

```
using System.Text;

public class AsynchIOTester
{
   private Stream inputStream;

   // делегируемый метод
   private AsyncCallback myCallBack;

   // буфер для хранения прочитанных данных
   private byte[] buffer;

   // размер буфера
   const int BufferSize = 256;

   // конструктор
   AsynchIOTester()
   {
      // открыть входной поток
      inputStream =
         File.OpenRead(
         @"C:\test\source\AskTim.txt");

      // выделить место в буфере
      buffer = new byte[BufferSize];

      // назначить обратный вызов
      myCallBack =
         new AsyncCallback(this.OnCompletedRead);
   }

   public static void Main()
   {
      // создать объект класса AsynchIOTester,
      // который будет вызывать конструктор
      AsynchIOTester theApp =
         new AsynchIOTester();

      // вызвать метод объекта
      theApp.Run();
   }

   void Run()
   {
      inputStream.BeginRead(
         buffer,             // куда помещать результат
         0,                  // смещение в буфере
         buffer.Length,      // размер буфера
         myCallBack,         // делегат обратного вызова
         null);              // локальный объект состояния

      // заняться чем-нибудь, пока выполняется чтение данных
      for (long i = 0; i < 500000; i++)
      {
         if (i%1000 == 0)
         {
```

```
                Console.WriteLine("i: {0}", i);
            }
        }

    }

    // метод обратного вызова
    void OnCompletedRead(IAsyncResult asyncResult)
    {
        int bytesRead =
            inputStream.EndRead(asyncResult);

        // если что-нибудь прочитано, преобразовать данные
        // в строковый тип, вывести их на экран и
        // начать все сначала;
        // в противном случае работа окончена
        if (bytesRead > 0)
        {
            String s =
                Encoding.ASCII.GetString(buffer, 0, bytesRead);
            Console.WriteLine(s);
            inputStream.BeginRead(
                buffer, 0, buffer.Length, myCallBack, null);
        }
    }
  }
}
```

Вывод (отрывок)

```
i: 47000
i: 48000
i: 49000
Date: January 2001
From: Dave Heisler
To: Ask Tim
Subject: Questions About O'Reilly
Dear Tim,
I've been a programmer for about ten years. I had heard of
O'Reilly books,then...
Dave,
You might be amazed at how many requests for help with
school projects I get;
i: 50000
i: 51000
i: 52000
```

Из вывода программы ясно, что она одновременно работала в двух вычислительных потоках. Чтение выполнялось в фоновом режиме, а основной поток увеличивал счетчик, выводя на экран каждое тысячное значение. Когда очередное чтение завершалось, полученные данные выводились на экран, после чего возобновлялась работа счетчика. (Здесь представлена лишь часть вывода программы.)

В реальном приложении во время асинхронного ввода/вывода в файл или базу данных можно обрабатывать пользовательские запросы или производить математические вычисления.

Сетевой ввод/вывод

Запись данных в удаленный объект в Интернете мало чем отличается от записи в файл на локальном компьютере. Необходимость в такой операции может возникнуть, когда программа сохраняет данные в файле, расположенном на другом компьютере локальной сети, или когда создается программа, выводящая информацию на экран другого компьютера.

Сетевой ввод/вывод основан на потоках, создаваемых с помощью сокетов. Сокеты оказываются очень полезными при создании приложений, работающих по схеме «клиент/сервер», по схеме P2P (peer-to-peer, равноправная связь) или при выполнении удаленных вызовов процедур.

Сокет – это объект, представляющий конечную точку коммуникации двух процессов, связанных между собой через сеть. Сокеты могут работать с различными протоколами, включая UDP и TCP/IP. В этом разделе будет создано соединение между клиентом и сервером по протоколу TCP/IP. Это протокол для сетевой коммуникации, основанный на соединениях. Слова «основанный на соединениях» означают, что с помощью этого протокола два процесса, установивших соединение, могут общаться так, словно они соединены прямой телефонной линией.

Хотя протокол TCP/IP разработан для общения через сеть, сетевое взаимодействие можно имитировать, запустив два процесса на одном компьютере.

Существует возможность одновременного общения нескольких приложений, работающих на одном компьютере, с несколькими разными клиентами (например, можно одновременно запустить веб-сервер, FTP-сервер и программу-калькулятор). Следовательно, каждое приложение должно иметь уникальный идентификационный номер, чтобы клиент мог указать, какое приложение ему нужно. Такой идентификационный номер называется *портом*. Можно представлять себе IP-адрес как телефонный номер, а порт – как добавочный номер.

Сервер создает экземпляр сокета и дает ему команду слушать соединения на конкретном порту. Конструктор сокета принимает один целочисленный аргумент, представляющий порт, прослушиваемый сокетом.

Приложения клиентов соединяются с определенным IP-адресом. Например, 216.114.108.245 – IP-адрес Yahoo. Кроме того, клиенты должны соединяться с определенным портом. По умолчанию все веб-броузеры соединяются с портом 80. Номера портов лежат в диапазоне от 0 до 65 535 (всего 2^{16}), впрочем, некоторые номера зарезервированы.

Порты делятся на несколько категорий:

- 0–1023 – известные порты
- 1024–49151 – зарегистрированные порты
- 49152–65535 – динамические и/или частные порты

Список всех известных и зарегистрированных портов можно узнать по адресу *http://www.iana.org/assignments/port-numbers*.

Если читатель работает в сети с брандмауэром, ему следует спросить у администратора сети, какие порты закрыты.

Создав сокет, программист должен вызвать его метод `Start()`, чтобы велеть сокету принимать сетевые соединения. Когда сервер готов отвечать на запросы клиентов, нужно вызвать метод `AcceptSocket()`. Выполнение потока, в котором вызван метод `AcceptSocket()`, приостанавливается (он печально сидит у виртуального телефона, надеясь, что ему позвонят).

Представим себе простейший сокет. Он ждет запрос от клиента и, когда запрос поступит, взаимодействует исключительно с этим клиентом. Все, кто позвонил после этого, автоматически ставятся в очередь. Пока они ждут («ваш звонок важен для нас, и он будет обслужен в порядке поступления»), работа их потоков приостанавливается. Когда журнал ждущих клиентов окажется заполненным, следующие клиенты будут получать в ответ на запрос некое подобие сигнала «занято». Они должны разъединиться и ждать, пока сокет обслужит текущего клиента. Подобная модель хорошо работает на серверах, принимающих один-два запроса в неделю, но она не справится с ситуациями, возникающими в реальном мире. Большинству серверов приходится обрабатывать тысячи, даже десятки тысяч соединений в минуту!

Чтобы обработать большое число соединений, приложения пользуются асинхронным вводом/выводом для принятия запроса и возвращения нового сокета, предоставляющего соединение с клиентом. После этого первый сокет возвращается к прослушиванию и ждет запроса от следующего клиента. Таким образом, программа может обработать много запросов: для каждого поступившего запроса создается новый сокет.

Клиент и не подозревает об этих «хитростях» с созданием новых сокетов. Ему достаточно того, что он соединен с сокетом, находящимся по нужному IP-адресу и на нужном порту. Обратите внимание, что сокет устанавливает с клиентом постоянное соединение. В этом его отличие от UDP, где используется протокол без соединения. В случае протокола TCP/IP клиент и сервер, установившие соединение друг с другом, знают, как общаться, и не нуждаются в повторной адресации каждого пакета данных.

Сам класс `Socket` достаточно прост. Он знает, как представлять конечную точку, но не умеет принимать запросы и создавать соединения

TCP/IP. Все это фактически выполняется классом TcpListener. Класс TcpListener надстраивается над классом Socket и реализует высокоуровневые сервисы TCP/IP.

Создание сетевого потокового сервера

Чтобы создать сетевой сервер для потока данных с протоколом TCP/IP, следует вначале создать объект TcpListener, который будет считывать информацию указанного порта TCP/IP. Среди доступных номеров портов произвольно выберем 65000:

```
TcpListener tcpListener = new TcpListener(65000);
```

Когда объект TcpListener будет создан, попросим его приступить к прослушиванию:

```
tcpListener.Start();
```

Будем ждать, пока какой-нибудь клиент не запросит соединение:

```
Socket socketForClient = tcpListener.AcceptSocket();
```

Метод AcceptSocket() объекта TcpListener возвращает объект Socket, представляющий *интерфейс сокетов Беркли* и связанный с конкретной конечной точкой. Метод AcceptSocket() — синхронный, и он не возвратит управление, пока не получит запрос на соединение.

Поскольку эта модель принята многими производителями компьютеров, *сокеты Беркли* упрощают задачу переноса существующих программ, использующих сокеты, как из среды Windows, так и из среды Unix.

Если соединение с сокетом установлено, можно посылать файл клиенту:

```
if (socketForClient.Connected)
{
```

Сначала создадим класс NetworkStream, передав сокет его конструктору:

```
NetworkStream networkStream = new NetworkStream(socketForClient);
```

Затем создадим объект StreamWriter примерно так же, как делали раньше, но сейчас не для файла, а для только что созданного объекта класса NetworkStream:

```
System.IO.StreamWriter streamWriter = new
System.IO.StreamWriter(networkStream);
```

Если выполнить вывод в этот поток данных, он будет по сети послан клиенту. Полный текст приложения сервера представлен в примере 21.8. (Автор до предела упростил его. В реальном сервере почти наверняка

присутствовал бы поток обработки запроса, а соответствующие выражения были бы помещены в блок `try` для обработки возможных сетевых проблем.)

Пример 21.8. Реализация сетевого потокового сервера

```
using System;
using System.Net.Sockets;

public class NetworkIOServer
{

   public static void Main()
   {

      NetworkIOServer app =
         new NetworkIOServer();
      app.Run();
   }

   private void Run()
   {
      // создать новый объект класса TcpListener и запустить
      // его на прослушивание порта 65000
      TcpListener tcpListener = new TcpListener(65000);
      tcpListener.Start();

      // продолжать прослушивание, пока не будет послан файл
      for (;;)
      {
         // по запросу клиента установить соединение и
         // возвратить новый сокет по имени socketForClient;
         // в это время tcpListener продолжает прослушивание
         Socket socketForClient = tcpListener.AcceptSocket();
         if (socketForClient.Connected)
         {
            Console.WriteLine("Client connected");

            // вызвать вспомогательный метод для отправки файла
            SendFileToClient(socketForClient);

            Console.WriteLine("Disconnecting from client...");

            // прибрать за собой и закончить работу
            socketForClient.Close();
            Console.WriteLine("Exiting...");
            break;
         }
      }
   }

   // вспомогательный метод для отправки файла
   private void SendFileToClient(
      Socket socketForClient )
   {
```

```
        // создать сетевой поток данных и объект, который пишет в него
        NetworkStream networkStream = new NetworkStream(socketForClient);
        System.IO.StreamWriter streamWriter =
           new System.IO.StreamWriter(networkStream);

        // создать объект, читающий файл из потока
        System.IO.StreamReader streamReader =
           new System.IO.StreamReader(@"C:\test\source\myTest.txt");

        string theString;

        // просмотреть строки файла в цикле, поочередно отсылая их клиенту

        do
        {
           theString = streamReader.ReadLine();

           if( theString != null )
           {
              Console.WriteLine(
                 "Sending {0}", theString);
              streamWriter.WriteLine(theString);
              streamWriter.Flush();
           }
        }
        while( theString != null );

        // прибрать за собой
        streamReader.Close();
        networkStream.Close();
        streamWriter.Close();
     }
  }
```

Создание сетевого потокового клиента

Клиент создает объект класса `TcpClient`, представляющего пользовательское соединение с хостом по протоколу TCP/IP:

```
TcpClient socketForServer;
socketForServer = new TcpClient("localHost", 65000);
```

С помощью этого класса можно создать объект `NetworkStream`, а для него – объект `StreamReader`:

```
NetworkStream networkStream = socketForServer.GetStream();
System.IO.StreamReader streamReader =
    new System.IO.StreamReader(networkStream);
```

Теперь будем читать из этого потока данные, пока они там есть, одновременно выводя их на экран:

```
do
{
```

```
        outputString = streamReader.ReadLine();

        if( outputString != null )
        {
            Console.WriteLine(outputString);
        }
    }
    while( outputString != null );
```

Полный текст программы клиента приведен в примере 21.9.

Пример 21.9. Реализация сетевого потокового клиента

```
using System;
using System.Net.Sockets;
public class Client
{

   static public void Main( string[] Args )
   {

      // создать объект TcpClient для общения с сервером
      TcpClient socketForServer;

      try
      {
         socketForServer = new TcpClient("localHost", 65000);
      }
      catch
      {
         Console.WriteLine(
            "Failed to connect to server at {0}:65000", "localhost");
         return;
      }

      // создать сетевой поток данных и объект, читающий его
      NetworkStream networkStream = socketForServer.GetStream();
      System.IO.StreamReader streamReader =
         new System.IO.StreamReader(networkStream);

      try
      {
         string outputString;

         // прочитать данные от хоста и вывести их на экран
         do
         {
            outputString = streamReader.ReadLine();

            if( outputString != null )
            {
               Console.WriteLine(outputString);
            }
         }
         while( outputString != null );
```

```
        }
        catch
        {
           Console.WriteLine("Exception reading from Server");
        }

        // прибрать за собой
        networkStream.Close();
     }
  }
```

Для тестирования этих программ напишем простой файл и назовем его *myText.txt*:

```
This is line one
This is line two
This is line three
This is line four
```

Вывод клиента и сервера будет следующий:

Вывод (на сервере):

```
Client connected
Sending This is line one
Sending This is line two
Sending This is line three
Sending This is line four
Disconnecting from client...
Exiting...
```

Вывод (на клиенте):

```
This is line one
This is line two
This is line three
This is line four
Press any key to continue
```

Если читатель тестирует этот код на одном компьютере, он должен запустить клиент и сервер в разных командных окнах или в разных экземплярах среды разработки. Сервер запускается первым, иначе клиент закончит свою работу аварийно из-за невозможности установить соединение.

Обработка нескольких соединений

Как было сказано ранее, этот пример не справится с повышенной нагрузкой, поскольку каждый клиент целиком поглощает внимание сервера. Необходим сервер, способный устанавливать соединение и затем передавать его программе, выполняющей асинхронный ввод/вывод, как это происходило при чтении файла.

Чтобы решить такую задачу, создадим новый сервер, `AsynchNetworkServer`, в котором будет создан новый класс, `ClientHandler`. Сервер

AsynchNetworkServer, приняв запрос клиента на соединение, создаст экземпляр класса ClientHandler и передаст ему сокет.

Конструктор ClientHandler() создаст копию сокета и буфер, а также откроет для сокета новый сетевой поток данных NetworkStream. После чего будет выполняться асинхронный ввод/вывод в этот сокет. Текст, полученный от клиента, будем простоты ради выводить на экран и отправлять обратно клиенту.

Чтобы реализовать асинхронный ввод/вывод, класс ClientHandler определяет два делегируемых метода, OnReadComplete() и OnWriteComplete(). Они будут управлять «перекрывающимся» вводом и выводом строк, посылаемых клиентом.

Тело метода Run() для сервера практически такое же, как и в примере 21.8. Сначала создается объект для прослушивания, а затем вызывается метод Start(). Потом пишется *бесконечный* цикл и вызывается метод AcceptSocket(). Когда сокет будет соединен, обработка соединения проводиться не будет. Вместо этого создается новый объект класса ClientHandler и вызывается его метод StartRead().

Полный исходный текст такого сервера содержится в примере 21.10.

Пример 21.10. Реализация асинхронного сетевого потокового сервера

```
using System;
using System.Net.Sockets;

public class AsynchNetworkServer
{

   class ClientHandler
   {
      public ClientHandler( Socket socketForClient )
      {
         socket = socketForClient;
         buffer = new byte[256];
         networkStream = new NetworkStream(socketForClient);

         callbackRead = new AsyncCallback(this.OnReadComplete);

         callbackWrite = new AsyncCallback(this.OnWriteComplete);
      }

      // начать чтение строки, переданной клиентом
      public void StartRead()
      {
         networkStream.BeginRead(
            buffer, 0, buffer.Length,
            callbackRead, null);
      }

      // получив обратный вызов, этот метод выводит строку
      // на экран и отсылает ее клиенту
      private void OnReadComplete( IAsyncResult ar )
```

```
    {
       int bytesRead = networkStream.EndRead(ar);

       if( bytesRead > 0 )
       {
          string s =
             System.Text.Encoding.ASCII.GetString(buffer, 0, bytesRead);
          Console.Write(
                "Received {0} bytes from client: {1}",
                 bytesRead, s );
          networkStream.BeginWrite(
             buffer, 0, bytesRead, callbackWrite, null);
       }
       else
       {
          Console.WriteLine( "Read connection dropped");
          networkStream.Close();
          socket.Close();
          networkStream = null;
          socket = null;
       }
    }

    // после записи строки вывести сообщение и возобновить чтение
    private void OnWriteComplete( IAsyncResult ar )
    {
       networkStream.EndWrite(ar);
       Console.WriteLine( "Write complete");
       networkStream.BeginRead(
          buffer, 0, buffer.Length,
          callbackRead, null);
    }

    private byte[]          buffer;
    private Socket          socket;
    private NetworkStream   networkStream;
    private AsyncCallback   callbackRead;
    private AsyncCallback   callbackWrite;
}

public static void Main()
{
   AsynchNetworkServer app = new AsynchNetworkServer();
   app.Run();
}

private void Run()
{
   // создать новый объект класса TcpListener и запустить
   // его на прослушивание порта 65000
   TcpListener tcpListener = new TcpListener(65000);
   tcpListener.Start();
```

```
        // продолжать прослушивание, пока не будет послан файл
        for (;;)
        {
           // по запросу клиента установить соединение и
           // возвратить новый сокет по имени socketForClient;
           // в это время tcpListener продолжает прослушивание
           Socket socketForClient = tcpListener.AcceptSocket();
           if (socketForClient.Connected)
           {
              Console.WriteLine("Client connected");
              ClientHandler handler = new ClientHandler(socketForClient);
              handler.StartRead();
          }
        }
     }
  }
```

После запуска сервер начинает слушать порт 65000. Когда клиент запросит соединение, сервер создаст экземпляр класса ClientHandler, который будет управлять вводом/выводом, пока сервер слушает, не появился ли запрос от следующего клиента.

В этом примере строка, полученная от клиента, выводится на экран методами OnReadComplete() и OnWriteComplete(). Вывод на экран может заблокировать выполнение потока до своего завершения. В реальном приложении наличие какой-либо блокировки в этих методах нежелательно, поскольку поток помещен в пул. Если в методе OnReadComplete() или OnWriteComplete() возникнет блокировка, в пуле потоков появятся новые потоки, что отрицательно скажется на производительности и способности сервера обрабатывать большое количество запросов.

Текст программы клиента весьма прост. Клиент создает сокет tcpSocket для порта, прослушиваемого сервером (65000), и объект класса NetworkStream для этого сокета. Затем он выводит сообщение в поток и освобождает буфер. Для чтения из потока клиент создает объект StreamReader. Прочитанные данные будут выводиться на экран. Полный текст программы клиента приведен в примере 21.11.

Пример 21.11. Реализация клиента для сетевого асинхронного ввода/вывода

```
using System;
using System.Net.Sockets;
using System.Threading;
using System.Runtime.Serialization.Formatters.Binary;

public class AsynchNetworkClient
{
   static public int Main()
```

```
    {
        AsynchNetworkClient client = new AsynchNetworkClient();
        return client.Run();
    }

    AsynchNetworkClient()
    {
        string serverName = "localhost";
        Console.WriteLine("Connecting to {0}", serverName);
        TcpClient tcpSocket = new TcpClient(serverName, 65000);
        streamToServer = tcpSocket.GetStream();
    }

    private int Run()
    {
        string message = "Hello Programming C#";
        Console.WriteLine(
            "Sending {0} to server.", message);

        // создать объект streamWriter и с его помощью
        // отправить строку на сервер
        System.IO.StreamWriter writer =
            new System.IO.StreamWriter(streamToServer);
        writer.WriteLine(message);
        writer.Flush();

        // прочитать ответ
        System.IO.StreamReader reader =
            new System.IO.StreamReader(streamToServer);
        string strResponse = reader.ReadLine();
        Console.WriteLine("Received: {0}", strResponse);
        streamToServer.Close();
        return 0;

    }

    private NetworkStream   streamToServer;
}
```

Вывод (на сервере):

```
Client connected
Received 22 bytes from client: Hello Programming C#
Write complete
Read connection dropped
```

Вывод (на клиенте):

```
Connecting to localhost
Sending Hello Programming C# to server.
Received: Hello Programming C#
```

В этом примере сетевой сервер не блокируется на время обработки запросов клиента. Вместо этого он делегирует управление соединениями экземплярам класса `ClientHandler`. Клиенты не будут испытывать задержку, ожидая, пока сервер обработает их запросы.

Асинхронные сетевые файловые потоки

В этом разделе знания, полученные вами об асинхронном чтении файлов, будут объединены со знаниями об асинхронных сетевых потоках. Результатом будет программа, посылающая клиенту файл в ответ на запрос.

Сервер начнет с асинхронного ввода из сокета – он будет ждать от клиента имя файла. Получив имя, можно будет запустить асинхронное чтение файла на сервере. После каждого заполнения буфера можно выполнять асинхронную отправку фрагментов файла клиенту. Когда эта отправка закончится, можно снова запускать чтение файла. Таким образом, программа переключается с чтения на запись и обратно, то заполняя буфер данными из файла, то отсылая буфер клиенту. Последнему ничего не надо делать, кроме чтения потока данных, идущих от сервера. В следующем примере клиент выводит содержимое файла на экран, но он с таким же успехом мог бы асинхронно записывать полученные данные в файл, реализуя копирование файлов через сеть.

Структура сервера мало отличается от той, что приведена в примере 21.10. Снова создается класс `ClientHandler`, но на этот раз добавляется объект типа `AsyncCallBack` по имени `myFileCallBack`. Он будет инициализирован в конструкторе наряду с объектами обратного вызова для сетевого чтения и записи:

```
myFileCallBack = new AsyncCallback(this.OnFileCompletedRead);

callbackRead = new AsyncCallback(this.OnReadComplete);

callbackWrite = new AsyncCallback(this.OnWriteComplete);
```

Функция `Run()` внешнего класса, теперь называемого `AsynchNetworkFileServer`, осталась без изменений. В ней по-прежнему создается и запускается класс `TcpListener` и организуется *бесконечный* цикл, в котором вызывается метод `AcceptSocket()`. При получении сокета создается экземпляр класса `ClientHandler` и вызывается метод `StartRead()`. Как и в предыдущем примере, этот метод вызывает `BeginRead()`, которому передает буфер и делегат `OnReadComplete()`.

Когда чтение сетевого потока данных завершается, вызывается делегированный метод `OnReadComplete()`, извлекающий имя файла из буфера. Если возвращен текст, метод `OnReadComplete()` считывает из буфера строку с помощью статического метода `System.Text.Encoding.ASCII.GetString()`:

```
if( bytesRead > 0 )
{
   string fileName =
      System.Text.Encoding.ASCII.GetString(
      buffer, 0, bytesRead);
```

Итак, имя файла получено. Теперь можно открывать поток данных к этому файлу и асинхронно читать его, как это делалось в примере 21.7.

```
inputStream = File.OpenRead(fileName);

inputStream.BeginRead(
   buffer,              // куда помещать результат
   0,                   // смещение в буфере
   buffer.Length,       // размер буфера
   myFileCallBack,      // делегат обратного вызова
   null);               // локальный объект состояния
```

Эта операция чтения файла имеет собственный метод обратного вызова, который будет вызван, когда входной поток данных заполнит буфер информацией, прочитанной из файла на сервере.

Как отмечалось ранее, предпринимать какие-либо действия в методе асинхронного ввода/вывода нежелательно, поскольку он может заблокировать рабочий поток на довольно значительное время. Вызовы методов открытия файла и начала чтения из него должны происходить во вспомогательном потоке, а не в методе `OnReadComplete()`. Данный пример упрощен, чтобы не отвлекать читателя от обсуждаемой темы.

Когда буфер заполнен, вызывается метод `OnFileCompletedRead()`, который проверяет, прочитаны ли данные из файла, и, если прочитаны, начинает асинхронный вывод в сеть:

```
if (bytesRead > 0)
{
   // переслать данные клиенту
   networkStream.BeginWrite(
      buffer, 0, bytesRead, callbackWrite, null);
}
```

Если метод `OnFileCompletedRead` был вызван, но ни одного байта не было получено, значит, пересылка файла завершена. В этом случае сервер закрывает поток `NetworkStream` и сокет, тем самым информируя клиента о завершении транзакции:

```
networkStream.Close();
socket.Close();
networkStream = null;
socket = null;
```

Когда вывод в сеть завершится, будет вызван метод `OnWriteComplete()`, который запустит новый раунд чтения файла:

```
private void OnWriteComplete( IAsyncResult ar )
{
```

```
      networkStream.EndWrite(ar);
      Console.WriteLine( "Write complete");

      inputStream.BeginRead(
         buffer,              // куда помещать результат
         0,                   // смещение в буфере
         buffer.Length,       // размер буфера
         myFileCallBack,      // делегат обратного вызова
         null);               // локальный объект состояния

   }
```

Цикл возобновляет чтение файла и продолжается, пока все содержимое не будет прочитано и отправлено клиенту. Что касается последнего, то он просто выводит имя файла в сеть и запускает операцию чтения файла:

```
   string message = @"C:\test\source\AskTim.txt";
   System.IO.StreamWriter writer =
       new System.IO.StreamWriter(streamToServer);
    writer.Write(message);
    writer.Flush();
```

Затем клиент входит в цикл, читая сетевой поток данных до тех пор, пока от сервера поступает информация. Закончив чтение и пересылку файла, сервер закроет сетевой поток. В начале программы-клиента логической переменной `fQuit` присваивается значение `false`, а для хранения данных, полученных от сервера, создается буфер:

```
   bool fQuit = false;
   while (!fQuit)
   {
      char[] buffer = new char[BufferSize];
```

Теперь можно создавать новый объект `StreamReader` по переменной `streamToServer` класса `NetworkStream`:

```
   System.IO.StreamReader reader =
      new System.IO.StreamReader(streamToServer);
```

Метод `Read()` принимает три аргумента: буфер, смещение начала чтения и размер буфера:

```
   int bytesRead = reader.Read(buffer,0, BufferSize);
```

Выполняется проверка, возвратил ли метод `Read()` какие-либо данные. Если результат отрицательный, значит, работа закончена, и логической переменной `fQuit` можно присваивать значение `true`, что приведет к выходу из цикла:

```
   if (bytesRead == 0)
      fQuit = true;
```

Если же число прочитанных байтов не равно нулю, данные, посланные сервером, можно вывести на экран или записать в файл, или обработать как-то иначе:

```
      else
      {
         string theString = new String(buffer);
         Console.WriteLine(theString);
      }
   }
```

После выхода из цикла нужно закрыть поток `NetworkStream`.

```
   streamToServer.Close();
```

Полный текст приложения сервера, снабженный подробными комментариями, содержится в примере 21.12, за которым следует пример 21.13 с текстом приложения клиента.

Пример 21.12. Реализация асинхронного сетевого файлового сервера

```
using System;
using System.Net.Sockets;
using System.Text;
using System.IO;

// получить от клиента имя файла
// открыть файл и переслать его содержимое
// от сервера клиенту
public class AsynchNetworkFileServer
{

   class ClientHandler
   {
      // конструктор
      public ClientHandler(
         Socket socketForClient )
      {
         // инициализировать переменную класса
         socket = socketForClient;

         // инициализировать буфер, предназначенный для
         // хранения содержимого файла
         buffer = new byte[256];

         // создать сетевой поток данных
         networkStream = new NetworkStream(socketForClient);

         // установить обратный вызов для операции
         // чтения файла
         myFileCallBack = new AsyncCallback(this.OnFileCompletedRead);

         // установить обратный вызов для операции
         // чтения из сетевого потока
         callbackRead = new AsyncCallback(this.OnReadComplete);
```

```
      // установить обратный вызов для операции
      // записи в сетевой поток
      callbackWrite = new AsyncCallback(this.OnWriteComplete);
   }

   // начать чтение строки, переданной клиентом
   public void StartRead(  )
   {
      // выполнить чтение из сети
      // получить имя файла
      networkStream.BeginRead(
         buffer, 0, buffer.Length,
         callbackRead, null);
   }

   // получив обратный вызов от операции чтения, вывести
   // строку на экран и отправить ее клиенту
   private void OnReadComplete( IAsyncResult ar )
   {
      int bytesRead = networkStream.EndRead(ar);

      // если получена строка
      if( bytesRead > 0 )
      {
         // преобразовать строку в имя файла
         string fileName =
            System.Text.Encoding.ASCII.GetString(
            buffer, 0, bytesRead);

         // вывести на экран
         Console.Write("Opening file {0}", fileName);

         // открыть входной поток данных файла
         inputStream = File.OpenRead(fileName);

         // начать считывать файл
         inputStream.BeginRead(
            buffer,             // куда помещать результат
            0,                  // смещение
            buffer.Length,      // размер буфера
            myFileCallBack,     // делегат обратного вызова
            null);              // локальный объект состояния

      }
      else

      {
         Console.WriteLine( "Read connection dropped");
         networkStream.Close(  );
         socket.Close(  );
         networkStream = null;
         socket = null;
      }
   }
```

```
   // если буфер заполнен данными из файла
   void OnFileCompletedRead(IAsyncResult asyncResult)
   {
      int bytesRead = inputStream.EndRead(asyncResult);

      // если из файла что-то прочитано…
      if (bytesRead > 0)
      {
         // …отправить это клиенту
         networkStream.BeginWrite(
            buffer, 0, bytesRead, callbackWrite, null);
      }
      else
      {
         Console.WriteLine("Finished.");
         networkStream.Close(  );
         socket.Close(  );
         networkStream = null;
         socket = null;
      }
   }

   // после отправки строки продолжить считывание файла
   private void OnWriteComplete( IAsyncResult ar )
   {
      networkStream.EndWrite(ar);
      Console.WriteLine( "Write complete");

      // начать чтение данных из файла
      inputStream.BeginRead(
         buffer,             // куда помещать результат
         0,                  // смещение
         buffer.Length,      // размер буфера
         myFileCallBack,     // делегат обратного вызова
         null);              // локальный объект состояния

   }

   private const int       BufferSize = 256;
   private byte[]          buffer;
   private Socket          socket;
   private NetworkStream   networkStream;
   private Stream          inputStream;
   private AsyncCallback   callbackRead;
   private AsyncCallback   callbackWrite;
   private AsyncCallback   myFileCallBack;

}

public static void Main(  )
{
   AsynchNetworkFileServer app =
```

```
            new AsynchNetworkFileServer(  );
         app.Run(  );
      }

      private void Run(  )
      {
         // содать новый TcpListener и запустить его
         // слушать на порту 65000
         TcpListener tcpListener = new TcpListener(65000);
         tcpListener.Start(  );

         // продолжать слушать до отправки файла
         for (;;)
         {
            // по запросу клиента установить соединение и
            // возвратить новый сокет по имени socketForClient;
            // в это время tcpListener продолжает слушать порт
            Socket socketForClient = tcpListener.AcceptSocket(  );
            if (socketForClient.Connected)
            {
               Console.WriteLine("Client connected");
               ClientHandler handler = new ClientHandler(socketForClient);
               handler.StartRead();
            }
         }
      }
   }
```

Пример 21.13. Реализация клиента для асинхронного сетевого файлового сервера

```
using System;
using System.Net.Sockets;
using System.Threading;
using System.Text;

public class AsynchNetworkClient
{

   static public int Main(  )
   {

      AsynchNetworkClient client = new AsynchNetworkClient(  );
      return client.Run( );
   }

   AsynchNetworkClient(  )
   {
      string serverName = "localhost";
      Console.WriteLine("Connecting to {0}", serverName);
      TcpClient tcpSocket = new TcpClient(serverName, 65000);
      streamToServer = tcpSocket.GetStream(  );
   }

   private int Run(  )
```

```
    {
       string message = @"C:\test\source\AskTim.txt";
       Console.Write("Sending {0} to server.", message);

       // создать объект streamWriter для отправки
       // строки серверу
       System.IO.StreamWriter writer =
            new System.IO.StreamWriter(streamToServer);
       writer.Write(message);
       writer.Flush( );

       bool fQuit = false;

       // продолжать чтение, пока поступают данные от сервера
       while (!fQuit)
       {
          // буфер для хранения ответа
          char[] buffer = new char[BufferSize];
          // прочитать ответ
          System.IO.StreamReader reader =
             new System.IO.StreamReader(streamToServer);

          // проверить, сколько байтов получено в буфер
          int bytesRead =
             reader.Read(buffer,0,BufferSize);
          if (bytesRead == 0)  // ни одного? закончить работу
             fQuit = true;
          else                 // что-то получено?
          {
             // вывести это на экран
             string theString = new String(buffer);
             Console.WriteLine(theString);
          }
       }
       streamToServer.Close(  ); // подчистить
       return 0;

    }

    private const int BufferSize = 256;
    private NetworkStream   streamToServer;
}
```

В результате сочетания асинхронного чтения файла с асинхронным чтением из сети получилось приложение, которое не только может обрабатывать запросы от нескольких клиентов, но и справляется с увеличением их числа.

Веб-потоки

Выполнять чтение любой веб-страницы в Интернете ничуть не сложнее, чем читать из потока данных, предоставленного сервером.

WebRequest — это объект, запрашивающий идентификатор URI, например URL-адрес веб-страницы. С помощью этого объекта можно создать объект WebResponse, который будет инкапсулировать объект, на который указывает идентификатор URI. Иными словами, можно вызвать метод GetResponse() объекта WebRequest, чтобы получить собственно объект, на который указывает URI (например, веб-страницу). Возвращаемый объект инкапсулируется в объекте WebResponse. После этого можно запросить у объекта WebResponse объект Stream, для чего следует вызвать метод GetResponseStream(). Этот метод возвратит поток данных, инкапсулирующий содержимое веб-объекта (например, это будет поток с веб-страницей).

В следующем примере содержимое веб-страницы читается именно в виде потока. Чтобы получить веб-страницу, нужен класс HttpWebRequest. Этот класс произведен от класса WebRequest; он поддерживает взаимодействие с протоколом HTTP.

Чтобы создать объект класса HttpWebRequest, следует выполнить преобразование типа объекта WebRequest, возвращенного статическим методом Create() класса WebRequest:

```
HttpWebRequest webRequest =
    (HttpWebRequest) WebRequest.Create
    ("http://www.libertyassociates.com/book_edit.htm");
```

Create() — это статический метод класса WebRequest. Если ему передать URI, он создаст экземпляр класса HttpWebRequest.

Этот метод перегружается по типу параметра и возвращает различные производные типы в зависимости от того, что ему передано. Например, если методу передать URI, будет создан объект типа HttpWebRequest. В то же время, возвращаемое значение имеет тип WebRequest. Вот почему необходимо преобразование возвращенного значения к типу HttpWebRequest.

Факт создания объекта HttpWebRequest устанавливает соединение с веб-страницей на сайте автора этой книги. То, что возвратит хост, будет инкапсулировано в объекте класса HttpWebResponse, представляющего собой подкласс более общего класса WebResponse и ориентированного на конкретный протокол HTTP:

```
HttpWebResponse webResponse = (HttpWebResponse) webRequest.GetResponse();
```

Теперь можно открыть поток данных StreamReader для этой страницы, вызвав метод GetResponseStream() объекта WebResponse:

```
StreamReader streamReader = new StreamReader(
    webResponse.GetResponseStream(), Encoding.ASCII);
```

Чтение из этого потока выполняется точно так же, как чтение из сетевого потока. Полный текст соответствующей программы приведен в примере 21.14.

Пример 21.14. Чтение веб-страницы в виде HTML-потока

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Text;

public class Client
{

   static public void Main( string[] Args )
   {

      // создать объект WebRequest для конкретной страницы
      HttpWebRequest webRequest =
         (HttpWebRequest) WebRequest.Create
         ("http://www.libertyassociates.com/book_edit.htm");

      // запросить у объекта WebRequest объект WebResponse,
      // инкапсулирующий эту страницу
      HttpWebResponse webResponse =
         (HttpWebResponse) webRequest.GetResponse();

      // получить объект streamReader от объекта WebResponse
      StreamReader streamReader = new StreamReader(
         webResponse.GetResponseStream(), Encoding.ASCII);

      try
      {
         string outputString;
         outputString = streamReader.ReadToEnd();
         Console.WriteLine(outputString);
      }
      catch
      {
         Console.WriteLine("Exception reading from web page");
      }
      streamReader.Close();
   }
}
```

Вывод (отрывок):

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
<title>Books &amp; Resources</title>
</head>

<body bgcolor="#ffffff" vlink="#808080"
```

```
alink="#800000" topmargin="0" leftmargin="0">
<table border="0" cellpadding="0" cellspacing="0" width="454"
bgcolor="#ffffff">

  <tr>
&quot;More
      than just about any other writer, Jesse Liberty
      is brilliant at communicating what it's really
      like to work on a programming project.&quot;
      </font></b><font face="times new roman, times,
      serif" size="3"><b>
 </b> Barnes &amp; Noble</font></i><font size="3"><br>
```

Как видно из выведенного текста, через поток данных переслан HTML-код запрошенной веб-страницы. Такой технологией можно воспользоваться для получения *копии экрана*, то есть для считывания страницы в буфер с последующим извлечением необходимой информации.

В этой книге во всех примерах, где используются копии экрана, подразумевается, что на эту операцию получено разрешение владельца авторских прав на сайт.

Сериализация

Когда объект направляется на диск в потоке данных, его различные элементы должны быть *сериализованы*, то есть выведены в поток в виде последовательности байтов. Объект сериализуется и в других случаях, таких как сохранение в базе данных или маршалинг для перехода через границы контекста, домена приложения, процесса или компьютера.

Среда CLR предоставляет поддержку сериализации в виде *объектного графа*, представляющего объект и все его элементы. Как было сказано в главе 19, по умолчанию ни один тип не сохраняется в потоке. Чтобы сохранить конкретный объект, для него необходимо явно установить атрибут `[Serializable]`.

В любом случае всю работу по сохранению объекта среда CLR берет на себя. Она знает, как сохранять базовые типы. Поэтому если объект состоит только из них (то есть все элементы имеют тип `int`, `long`, `string` и т. д.), то программисту вообще ничего не надо делать. Если же объект содержит типы, определяемые пользователем (классы), то он должен их тоже сделать сериализуемыми. Среда CLR попытается сохранить каждый объект, содержащийся в сериализуемом объекте (и все объекты, содержащиеся в них), поэтому все внутренние объекты должны либо иметь базовый тип, либо быть сериализуемыми.

Это было очевидно уже в главе 19, где для маршалинга создавался объект `Shape`, содержащий в качестве своего члена объект `Point`. Последний, в свою очередь, содержал элементы базовых типов. Чтобы се-

риализовать (и тем самым подготовить к маршалингу) объект Shape, пришлось пометить его объект Point как сериализуемый.

Когда для объекта выполняется маршалинг по ссылке или по значению, объект должен быть сериализован. Разница в способах передачи сводится к тому, что может либо создаваться копия объекта, либо клиенту может предоставляться заместитель. Объекты, помеченные атрибутом [Serializable], маршалируются по значению, а объекты класса, производного от MarshalByRefObject, маршалируются по ссылке. Однако сериализуются и те и другие. Подробности см. в главе 19.

Применение форматизатора

Сериализованные данные в конце концов будут прочитаны либо той же самой программой, либо программой, работающей на другом компьютере. Как бы то ни было, код, читающий данные, ожидает, что они имеют определенный формат. В большинстве .NET-приложений таким форматом будет либо исходный двоичный формат, либо SOAP.

SOAP (Simple Object Access Protocol, простой протокол доступа к объектам) представляет собой облегченный протокол обмена информацией в Сети, в основе которого лежит стандарт XML. Протокол SOAP в значительной степени является модульным и расширяемым. Он также способствует применению таких Интернет-технологий, как HTTP и SMTP.

Формат сериализации данных определяется тем, какой *форматизатор* указан программистом. В главе 19 форматизаторы использовались совместно с каналами при взаимодействии с удаленным объектом. Классы форматизаторов реализуют интерфейс Iformatter. Программист может создать и собственный форматизатор, но это мало кому нужно, да и не каждый возьмется за такую работу! Среда CLR предоставляет разработчикам как класс SoapFormatter для сериализации в Интернете, так и BinaryFormatter, полезный при сохранении объектов на локальном диске.

Экземпляры этих объектов создаются с помощью конструкторов, предоставляемых по умолчанию:

```
BinaryFormatter binaryFormatter = New BinaryFormatter();
```

Получив объект форматизатора, программист вызывает метод Serialize(), передавая ему поток и сериализуемый объект. В следующем примере будет показано, как это делается.

Выполнение сериализации

Чтобы продемонстрировать сериализацию в работе, понадобится демонстрационный класс, который можно будет сохранить в потоке, а затем считать оттуда его состояние. Начнем с создания класса SumOf, у которого будет три переменных:

```
private int startNumber = 1;
private int endNumber;
private int[] theSums;
```

Массив theSums хранит суммы всех чисел от startNumber до endNumber. Так, если startNumber равно 1, а endNumber равно 10, то массив содержит следующие значения:

```
1, 3, 6, 10, 15, 21, 28, 36, 45, 55
```

Каждое значение равно сумме предыдущего и следующего чисел из последовательности. Так, в последовательности 1,2,3,4 первое значение равно 1. Второе – сумме предыдущего (1) и следующего чисел из последовательности (2). Иными словами, theSums[1] содержит 3. Аналогичным образом третье значение равно сумме второго (3) и следующего чисел из последовательности (3), то есть theSums[2] равно 6. Наконец, четвертое значение в массиве theSums будет суммой третьего (6) и числа 4, то есть числом 10.

Конструктор объекта SumOf принимает два целых числа, начальное и конечное. Он присваивает их локальным переменным и вызывает вспомогательную функцию, вычисляющую содержимое массива:

```
public SumOf(int start, int end)
{
    startNumber = start;
    endNumber = end;
    ComputeSums();
```

Эта функция заполняет массив, вычисляя суммы в последовательности от startNumber до endNumber включительно:

```
private void ComputeSums()
{
    int count = endNumber - startNumber + 1;
    theSums = new int[count];
    theSums[0] = startNumber;
    for (int i=1,j=startNumber + 1;i<count;i++,j++)
    {
        theSums[i] =  j + theSums[i-1];
    }
}
```

Содержимое массива можно в любой момент вывести на экран в цикле foreach:

```
private void DisplaySums()
{
    foreach(int i in theSums)
    {
        Console.WriteLine("{0}, ",i);
    }
}
```

Сохранение объекта

Атрибутом [Serializable] пометим класс как подлежащий сериализации:

```
[Serializable]
class SumOf
```

Чтобы выполнить сериализацию, нужен файловый поток, в который будет сериализован объект SumOf:

```
FileStream fileStream = new FileStream("DoSum.out",FileMode.Create);
```

Теперь все готово к вызову метода Serialize() выбранного форматизатора и передачи ему потока и сериализуемого объекта. Поскольку все это делается в методе объекта SumOf, вторым аргументом является ключевое слово this, указывающее на текущий объект:

```
binaryFormatter.Serialize(fileStream,this);
```

Этот оператор приведет к сохранению объекта на диске.

Восстановление объекта

Чтобы восстановить объект, откройте файл и попросите двоичный форматизатор восстановить его:

```
public static SumOf DeSerialize()
{
    FileStream fileStream = new FileStream("DoSum.out",FileMode.Open);
     BinaryFormatter binaryFormatter = new BinaryFormatter();
    return (SumOf) binaryFormatter.Deserialize(fileStream);
    fileStream.Close();
}
```

Проверим, как все это работает. Сначала создадим новый объект типа SumOf и велим ему сериализовать самого себя. Потом создадим новый объект класса SumOf, установим в нем сохраненное состояние объекта и выведем значения массива:

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

```
        Console.WriteLine("Creating first one with new...");
        SumOf app = new SumOf(1,10);

        Console.WriteLine("Creating second one with deserialize...");
        SumOf newInstance = SumOf.DeSerialize();
        newInstance.DisplaySums();
    }
```

В примере 21.15 представлен полный исходный текст приложения, иллюстрирующий сохранение и восстановление объектов.

Пример 21.15. Сохранение и восстановление объекта

```
namespace Programming_CSharp
{
   using System;
   using System.IO;
   using System.Runtime.Serialization;
   using System.Runtime.Serialization.Formatters.Binary;

   [Serializable]
   class SumOf
   {

      public static void Main()
      {
         Console.WriteLine("Creating first one with new...");
         SumOf app = new SumOf(1,10);

         Console.WriteLine("Creating second one with deserialize...");
         SumOf newInstance = SumOf.DeSerialize();
         newInstance.DisplaySums();
      }

      public SumOf(int start, int end)
      {
         startNumber = start;
         endNumber = end;
         ComputeSums();
         DisplaySums();
         Serialize();
      }

      private void ComputeSums()
      {
         int count = endNumber - startNumber + 1;
         theSums = new int[count];
         theSums[0] = startNumber;
         for (int i=1,j=startNumber + 1;i<count;i++,j++)
         {
            theSums[i] =  j + theSums[i-1];

         }
      }
```

```
        private void DisplaySums()
        {
           foreach(int i in theSums)
           {
              Console.WriteLine("{0}, ",i);
           }
        }

        private void Serialize()
        {
           Console.Write("Serializing...");
           // создать файловый поток для записи
           FileStream fileStream = new FileStream("DoSum.out",FileMode.Create);
           // воспользоваться двоичным форматизатором, предоставленным средой CLR
           BinaryFormatter binaryFormatter = new BinaryFormatter();
           // сохранить объект на диске
           binaryFormatter.Serialize(fileStream,this);
           Console.WriteLine("...completed");
           fileStream.Close();
        }

        public static SumOf DeSerialize()
        {
           FileStream fileStream = new FileStream("DoSum.out",FileMode.Open);
           BinaryFormatter binaryFormatter = new BinaryFormatter();
           return (SumOf) binaryFormatter.Deserialize(fileStream);
           fileStream.Close();

        }
        private int startNumber = 1;
        private int endNumber;
        private int[] theSums;
    }
}
```

Вывод:

```
Creating first one with new...
1,
3,
6,
10,
15,
21,
28,
36,
45,
55,
Serializing......completed
Creating second one with deserialize...
1,
3,
6,
10,
```

```
15,
21,
28,
36,
45,
55,
```

Из выведенного текста ясно, что объект был создан, выведен на экран и сохранен в файле. После этого он был восстановлен и снова выведен на экран, причем без потери данных.

Обработка временных данных

В некоторых случаях подход к сериализации, продемонстрированный в примере 21.15, оказывается весьма неэкономичным. Поскольку содержимое массива легко вычисляется по начальному и конечному значению, нет веских причин хранить элементы массива на диске. Хотя сериализация маленького массива не требует больших затрат, она может оказаться довольно дорогостоящей операцией на больших массивах.

Программист может сообщить сериализатору, что некоторые данные сохранять не надо. Соответствующие элементы следует пометить атрибутом `[NonSerialized]`:

```
[NonSerialized] private int[] theSums;
```

Необходимо заметить, что если массив не будет сохранен, то при восстановлении объекта результат окажется не вполне корректным. Массив будет пуст. Ведь при восстановлении объекта он просто преобразуется из одной формы в другую, и никакие методы на вызываются.

Чтобы полностью восстановить объект перед возвращением его вызвавшему объекту, реализуем интерфейс `IDeserializationCallback`:

```
[Serializable]
class SumOf : IDeserializationCallback
```

Реализуем один метод этого интерфейса, а именно `OnDeserialization()`. Среда CLR гарантирует программисту, что если он реализует этот интерфейс, то она вызовет метод `OnDeserialization()` по окончании восстановления всего объектного графа. Как раз это и требуется: CLR восстановит то, что было сохранено, и даст программисту возможность восстановить несохраненные части.

Реализация интерфейса будет совсем простой: попросим объект заполнить массив:

```
public virtual void OnDeserialization (Object sender)
{
    ComputeSums();
}
```

Здесь налицо классическая проблема поиска компромисса между затратами памяти и времени. Если массив не сохраняется, процесс восстановления замедляется (поскольку приходится вычислять элементы массива), однако файл занимает меньше места на диске. Чтобы проверить, насколько эффективен отказ от сохранения массива, автор выполнил программу с числами от 1 до 5000. До постановки атрибута [NonSerialized] перед массивом сохраненный файл занимал 20 Кбайт. После постановки этого атрибута он уменьшился до 1 Кбайт. Неплохо. В примере 21.16 приведен исходный текст программы, в котором берется последовательность от 1 до 5 (чтобы сократить вывод).

Пример 21.16. Работа с несериализованным объектом

```
namespace Programming_CSharp
{
   using System;
   using System.IO;
   using System.Runtime.Serialization;
   using System.Runtime.Serialization.Formatters.Binary;

   [Serializable]
   class SumOf : IDeserializationCallback
   {

      public static void Main()
      {
         Console.WriteLine("Creating first one with new...");
         SumOf app = new SumOf(1,5);

         Console.WriteLine("Creating second one with deserialize...");
         SumOf newInstance = SumOf.DeSerialize();
         newInstance.DisplaySums();
      }

      public SumOf(int start, int end)
      {
         startNumber = start;
         endNumber = end;
         ComputeSums();
         DisplaySums();
         Serialize();
      }

      private void ComputeSums()
      {
         int count = endNumber - startNumber + 1;
         theSums = new int[count];
         theSums[0] = startNumber;
         for (int i=1,j=startNumber + 1;i<count;i++,j++)
         {
            theSums[i] =  j + theSums[i-1];

         }
      }
```

```
        private void DisplaySums()
        {
           foreach(int i in theSums)
           {
              Console.WriteLine("{0}, ",i);
           }
        }

        private void Serialize()
        {
           Console.Write("Serializing...");
           // создать файловый поток для записи
           FileStream fileStream = new FileStream("DoSum.out",FileMode.Create);
           // воспользоваться двоичным форматизатором, предоставленным средой CLR
           BinaryFormatter binaryFormatter = new BinaryFormatter();
           // сохранить объект на диске
           binaryFormatter.Serialize(fileStream,this);
           Console.WriteLine("...completed");
           fileStream.Close();
        }

        public static SumOf DeSerialize()
        {
           FileStream fileStream = new FileStream("DoSum.out",FileMode.Open);
           BinaryFormatter binaryFormatter = new BinaryFormatter();
           return (SumOf) binaryFormatter.Deserialize(fileStream);
           fileStream.Close();

        }

        // восстановить несохраненную информацию
        public virtual void OnDeserialization
           (Object sender)
        {
           ComputeSums();
        }

        private int startNumber = 1;
        private int endNumber;
        [NonSerialized] private int[] theSums;
    }
}
```

Вывод:

```
Creating first one with new...
1,
3,
6,
10,
15,
Serializing......completed
Creating second one with deserialize...
1,
```

```
3,
6,
10,
15,
```

Как видно из выведенного текста, данные были успешно сохранены на диске и затем восстановлены. Экономия места на диске за счет замедления работы практически незаметна при пяти значениях, однако она начнет играть существенную роль, когда число элементов массива дойдет до пяти миллионов.

До сих пор данные посылались в поток на диск для хранения и в сеть для упрощения связи с удаленными программами. Существует еще одна ситуация, в которой требуется создавать поток данных: необходимость долговременного хранения данных о конфигурации и состоянии, относящихся к отдельным пользователям. Для этой цели платформа .NET Framework предоставляет механизм, называемый *изолированной памятью*.

Изолированная память

Среда .NET CLR предоставляет механизм *изолированной памяти* (*Isolated Storage*), позволяющий программисту хранить информацию *отдельно по каждому пользователю*. Изолированная память во многом напоминает традиционные *.ini*-файлы Windows или более поздний ключ `HKEY_CURRENT_USER` в реестре Windows.

Приложения сохраняют данные в уникальных *отсеках* (*data compartments*), связанных с этими приложениями. Среда CLR реализует отсеки данных с помощью *хранилищ* (*data stores*), как правило, это каталоги файловой системы.

Администраторы вправе ограничивать размер изолированной памяти, предоставляемой отдельным приложениям. Они также вправе устанавливать защиту, не позволяющую ненадежному коду вызывать более надежный код для записи в изолированную память.

Важно, что для хранения данных в изолированной памяти среда CLR отводит стандартное место, но она не выдвигает никаких требований относительно формата этих данных. Короче говоря, в изолированной памяти можно хранить все, что угодно.

В большинстве случаев там хранится текст, часто в форме пар «имя/значение». Изолированная память представляет собой хорошее средство сохранения информации о пользовательской конфигурации, то есть имени для входа в систему, координат окон и элементов управления, а также сведений, специфичных для конкретного пользователя и конкретного приложения. Данные по каждому пользователю хранятся в отдельных файлах, но степень изоляции можно повысить еще больше, если проводить различие между отдельными аспектами иден-

тификации кода (по сборке или по домену приложения, из которого исходит машинный код).

Использование изолированной памяти не таит в себе никаких сложностей. Чтобы записать в нее информацию, надо создать экземпляр класса `IsolatedStorageFileStream` и инициализировать его именем файла и режимом доступа («создать», «присоединить» и т. д.):

```
IsolatedStorageFileStream configFile =
    new IsolatedStorageFileStream
    ("Tester.cfg",FileMode.Create);
```

Далее для этого файла создается объект `StreamWriter`:

```
StreamWriter writer = new StreamWriter(configFile);
```

Теперь можно выводить данные в этот поток, как в любой другой. Иллюстрация сказанного приведена в примере 21.17.

Пример 21.17. Запись в изолированную память

```
namespace Programming_CSharp
{
    using System;
    using System.IO;
    using System.IO.IsolatedStorage;

    public class Tester
    {

        public static void Main()
        {
            Tester app = new Tester();
            app.Run();
        }

        private void Run()
        {
            // создать поток для файла конфигурации
            IsolatedStorageFileStream configFile =
                new IsolatedStorageFileStream
                ("Tester.cfg",FileMode.Create);

            // создать объект, пишущий в этот поток
            StreamWriter writer = new StreamWriter(configFile);

            // вывести данные в файл конфигурации
            String output;
            System.DateTime currentTime = System.DateTime.Now;
            output = "Last access: " + currentTime.ToString();
            writer.WriteLine(output);
            output = "Last position = 27,35";
            writer.WriteLine(output);

            // освободить буфер и закончить работу
            writer.Flush();
```

```
            writer.Close();
            configFile.Close();
        }
    }
}
```

После выполнения этой программы поищите на диске файл *test.cfg*. На компьютере автора он появится в каталоге:

```
c:\Documents and Settings\Administrator\ApplicationData\
Microsoft\COMPlus\IsolatedStorage\0.4\
Url.wj4zpd5ni41dynqxx1uz0x0aoaraftc\
Url.wj4zpd5ni41dynqxx1uz0ix0aoaraftc\files
```

Если файл содержит только текст, его можно прочитать в редакторе Блокнот:

```
Last access: 5/2/2001 10:00:57 AM
Last position = 27,35
```

К этим данным можно обратиться и из программы. Для этого снова откроем файл:

```
IsolatedStorageFileStream configFile =
    new IsolatedStorageFileStream
    ("Tester.cfg",FileMode.Open);
```

Создадим объект `StreamReader`:

```
StreamReader reader = new StreamReader(configFile);
```

и воспользуемся стандартной потоковой идиомой чтения файла:

```
string theEntry;
do
{
    theEntry = reader.ReadLine();
    Console.WriteLine(theEntry);
} while (theEntry != null);
Console.WriteLine(theEntry);
```

Изолированная память существует в пределах сборки, то есть после перезапуска программы можно будет обратиться к тому же конфигурационному файлу. Пример 21.18 демонстрирует метод, необходимый для чтения изолированной памяти. Замените метод `Run()` предыдущего примера, перекомпилируйте и запустите его (но не меняйте его имя, иначе доступ к созданной ранее изолированной памяти будет утерян):

Пример 21.18. Чтение данных из изолированной памяти

```
        private void Run(  )
        {
            // открыть поток конфигурационного файла
```

```
            IsolatedStorageFileStream configFile =
                new IsolatedStorageFileStream
                ("Tester.cfg",FileMode.Open);

            // создать стандартный объект, читающий из потока
            StreamReader reader = new StreamReader(configFile);

            // считать файл и отобразить его
            string theEntry;
            do
            {
                theEntry = reader.ReadLine(  );
                Console.WriteLine(theEntry);
            } while (theEntry != null);

            reader.Close(  );
            configFile.Close(  );
        }
```

Вывод:

```
Last access: 5/2/2001 10:00:57 AM
Last position = 27,35
```

22

Взаимодействие .NET и COM

У каждого программиста чистые листы бумаги вызывают самые теплые чувства. Как было бы хорошо выкинуть все написанные нами программы и начать все с начала! К сожалению, никакая фирма на это не согласится. За прошедшее десятилетие многие фирмы-разработчики программных продуктов вложили большие средства в покупку компонентов COM и элементов управления ActiveX. Если .NET стремится стать жизнеспособной платформой, ее приложения обязаны уметь обращаться с этими компонентами, доставшимися им в наследство. Кроме того, компоненты .NET должны быть доступны в компонентах COM, хотя это требование и не такое важное, как первое.

В этой главе описывается, как платформа .NET поддерживает импорт элементов управления ActiveX и компонентов COM в пользовательские приложения и экспорт классов .NET в приложения, использующие модель COM, а также то, как в ней реализованы непосредственные вызовы функций API Win32. Кроме того, читатель узнает об указателях C# и ключевых словах, предоставляющих прямой доступ к памяти (технология, способная разрушить некоторые приложения).

Импорт элементов управления ActiveX

Элементы управления ActiveX – это COM-компоненты, обычно объединенные в форму, которые могут иметь (а могут и не иметь) пользовательский интерфейс. Когда корпорация Microsoft представила стандарт OCX, позволивший разработчикам строить элементы управления на языке Visual Basic и пользоваться ими в программах на C++ (и наоборот), произошла революция. За последние несколько лет тысячи таких элементов управления были разработаны и проданы. Они неве-

лики и удобны в обращении, так что представляют собой прекрасный пример многократно используемого кода.

Импорт элементов управления ActiveX в среду .NET на удивление прост, принимая во внимание несовместимость двоичных стандартов COM и .NET. Инструментальная среда Visual Studio .NET в состоянии импортировать элементы ActiveX, а кроме того, в Microsoft разработана утилита AxImp, которую можно вызывать в командной строке. Эта программа создает сборки, необходимые для использования элемента ActiveX в приложении .NET.

Создание элемента управления ActiveX

Для демонстрации возможностей применения классических элементов управления ActiveX в приложении .NET создадим элемент управления ActiveX, представляющий собой калькулятор, выполняющий четыре действия, и вызовем его из приложения, написанного на языке C#. Элемент управления ActiveX будет написан на языке VB6 и протестирован в VB6-приложении. Если у читателя нет среды VB6 или он не хочет строить элемент управления, готовый калькулятор можно загрузить с веб-сайта автора этой книги (*http://www.LibertyAssociates.com*).

Проверив работу элемента управления в стандартном окружении Windows, можно скопировать его в среду разработки .NET, зарегистрировать и импортировать в приложение Windows Forms.

Чтобы создать элемент управления, откройте VB6 и выберите ActiveX Control в качестве типа нового проекта. Форму проекта сделайте как можно меньше, поскольку у этого элемента управления не будет пользовательского интерфейса. Щелкните правой кнопкой мыши по элементу управления UserControl1 и выберите команду Properties в появившемся контекстном меню. В окне Properties переименуйте элемент управления в Calculator. Щелкните по узлу Project в окне проекта и в окне Properties укажите новое имя – CalcControl. Сразу после этого сохраните проект и дайте имя CalcControl как файлу, так и проекту (рис. 22.1).

Теперь добавьте в проект четыре функции. Для этого щелкните правой кнопкой мыши по форме CalcControl, выберите команду View Code в контекстном меню и введите текст программы на языке Visual Basic, представленный в примере 22.1.

Пример 22.1. Реализация элемента управления ActiveX по имени CalcControl

```
Public Function Add(left As Double, right As Double) As Double
    Add = left + right
End Function

Public Function Subtract(left As Double, right As Double) As Double
    Subtract = left - right
End Function
```

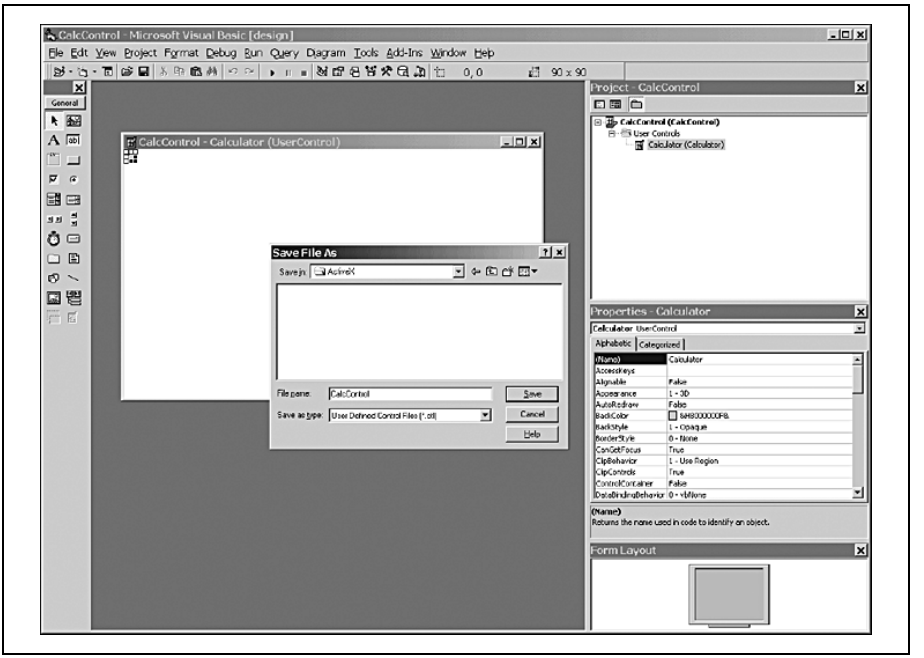

Рис. 22.1. *Создание элемента управления ActiveX в среде VB*

```
Public Function Multiply(left As Double, right As Double) As Double
    Multiply = left * right
End Function

Public Function Divide(left As Double, right As Double) As Double
    Divide = left / right
End Function
```

Это полный текст программы элемента управления. Скомпилируйте его в файл *CalcControl.ocx*, выполнив команду File → Make CalcControl.ocx меню Visual Basic 6.

Теперь откроем в среде VB второй проект как стандартный исполняемый (EXE). Назовем форму `TestForm`, а сам проект – `CalcTest`. Сохраните файл и проект под именем `CalcTest`.

Для того чтобы добавить элемент управления ActiveX в качестве компонента, нажмите комбинацию клавиш <Cntrl>+<T> и выделите элемент управления CalcControl на вкладке Controls, изображенной на рис. 22.2.

В результате в окне Toolbox появится новый элемент управления, обведенный кружком на рис. 22.3.

Перетащите этот элемент на форму `TestForm` и назовите его CalcControl. Обратите внимание, что новый элемент управления не будет виден, поскольку не имеет пользовательского интерфейса. Добавьте на форму два текстовых поля, четыре кнопки и статический текст, как показано на рис. 22.4.

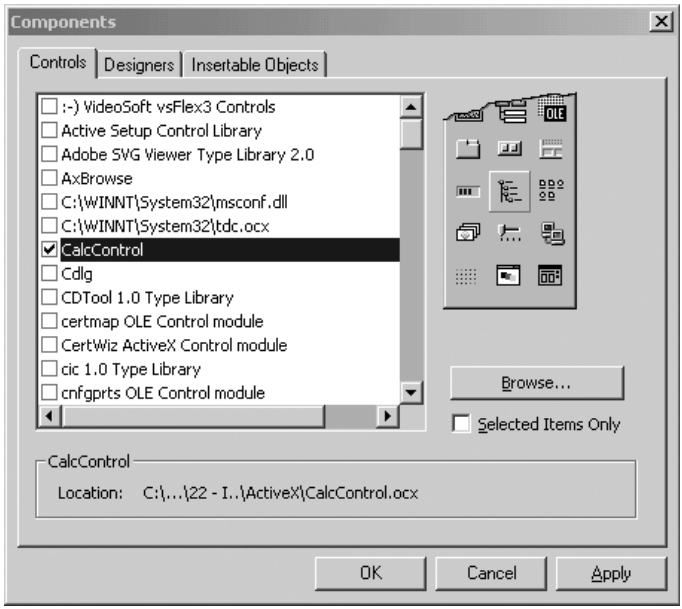

Рис. 22.2. *Добавление элемента управления CalcControl в панель инструментов VB6*

Рис. 22.3. *Местонахождение элемента управления CalcControl на панели инструментов VB6*

Назовите кнопки именами `btnAdd`, `btnSubtract`, `btnMultiply` и `btnDivide`. Теперь остается лишь реализовать методы, обрабатывающие щелчки по кнопкам. Каждый раз, когда пользователь щелкает по любой из кнопок, нужно прочитать значения в текстовых полях, привести их к типу `double` (как требует того элемент `CalcControl`) с помощью функции CDbl языка Visual Basic 6, вызвать функцию CalcControl и вывести результаты в поле метки. Текст выполняющей эти операции программы приведен в примере 22.2.

Пример 22.2. Тестирование элемента управления ActiveX с именем CalcControl в программе на языке VB (TestForm)

```
Private Sub btnAdd_Click()
    Label1.Caption = calcControl.Add(CDbl(Text1.Text), CDbl(Text2.Text))
```

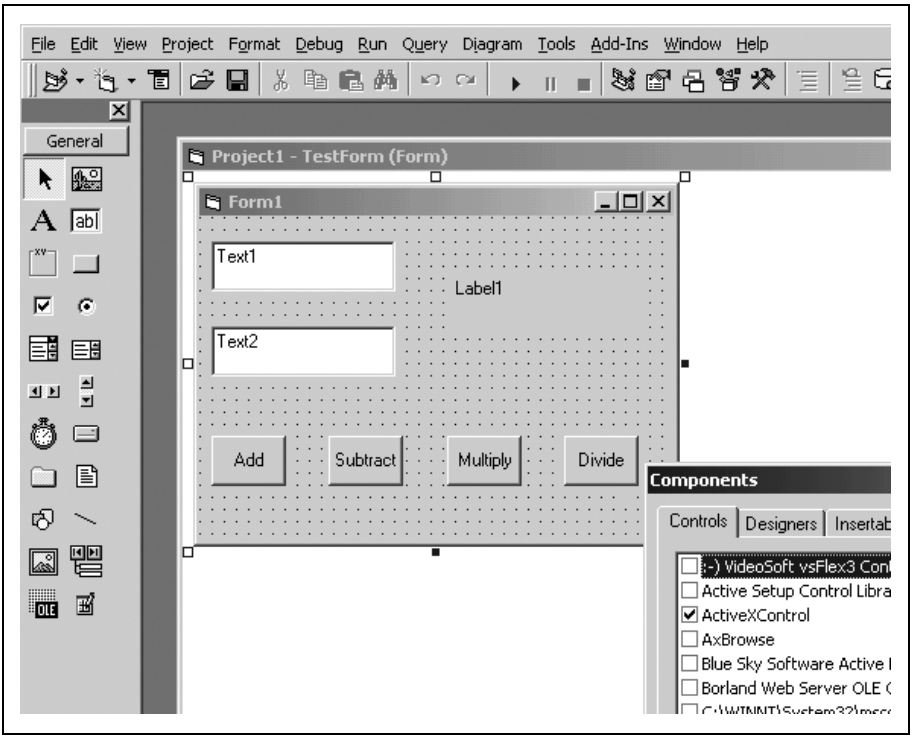

Рис. 22.4. *Создание пользовательского интерфейса формы TestForm*

```
End Sub

Private Sub btnDivide_Click()
    Label1.Caption = calcControl.Divide(CDbl(Text1.Text), CDbl(Text2.Text))
End Sub

Private Sub btnMultiply_Click()
    Label1.Caption = calcControl.Multiply(CDbl(Text1.Text), CDbl(Text2.Text))
End Sub

Private Sub btnSubtract_Click()
    Label1.Caption = calcControl.Subtract(CDbl(Text1.Text), CDbl(Text2.Text))
End Sub
```

На рис. 22.5 показан результат работы программы `CalcTest` после того, как были введены два числа и была нажата кнопка Multiply.

Импортирование элемента управления в платформу .NET

Теперь, когда элемент управления CalcControl проверен в деле, можно копировать файл *CalcControl.ocx* в инструментальную среду .NET. После копирования следует зарегистрировать файл с помощью утилиты

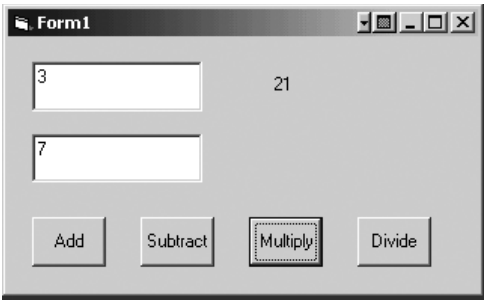

Рис. 22.5. *Результат работы программы CalcTest*

Regsvr32, и можно будет приступать к созданию тестовой программы, использующей калькулятор в платформе .NET:

```
Regsvr32 CalcControl.ocx
```

Для начала создадим в среде Visual Studio .NET проект C# Windows Form (см. главу 13), назовем его InteropTest и изобразим форму, подобную той, что была создана в среде Visual Basic в предыдущем разделе. Для этого перетащим на форму требуемые элементы, а саму форму тоже назовем TestForm. Примерный вид формы показан на рис. 22.6.

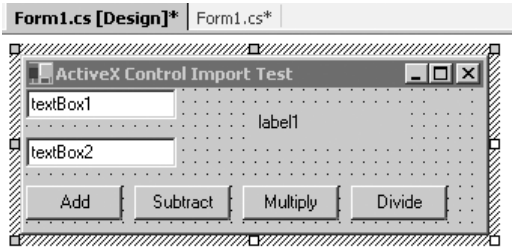

Рис. 22.6. *Windows-форма, тестирующая элемент управления ActiveX по имени CalcControl*

Импортирование элемента управления

Существует два способа импортировать элемент управления ActiveX в инструментальную среду Visual Studio .NET. Можно воспользоваться инструментами, имеющимися в среде, а можно выполнить импорт вручную, с помощью утилиты *aximp*, которая входит в комплект .NET Framework SDK. В среде Visual Studio .NET выполните команду меню Tools → Customize Toolbox. На вкладке COM Components найдите только что зарегистрированный объект CalcControl.Calculator (рис. 22.7).

Поскольку объект CalcControl зарегистрирован на платформе .NET, среда Visual Studio .NET легко найдет его. Достаточно выделить этот

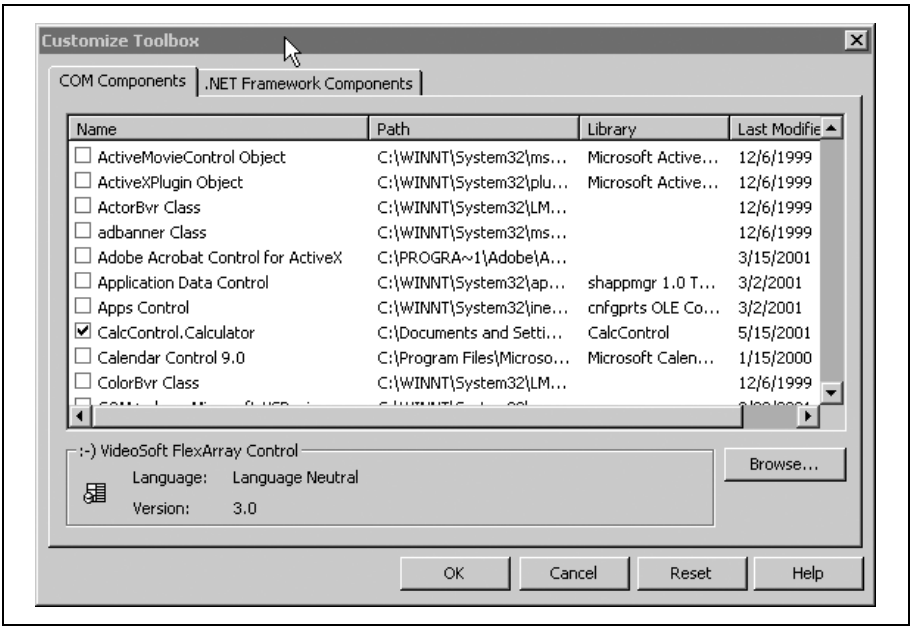

Рис. 22.7. *Импорт элемента управления ActiveX по имени CalcControl*

элемент управления в диалоговом окне Customize Toolbox, и он будет импортирован в приложение. Среда Visual Studio .NET сама позаботится о деталях. В качестве альтернативы можно открыть окно командной строки и импортировать элемент управления вручную с помощью утилиты *aximp.exe*, как показано на рис. 22.8.

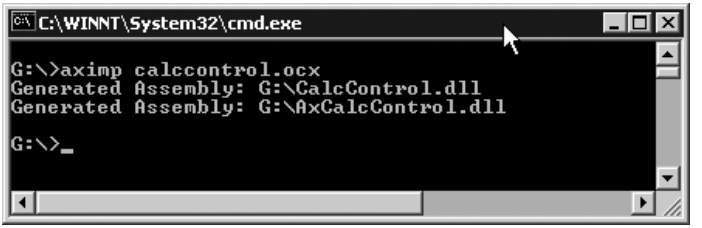

Рис. 22.8. *Запуск утилиты aximp*

Утилита *aximp.exe* имеет один аргумент – элемент управления ActiveX, который нужно импортировать (в данном случае *CalcControl.dll*). Она создает три файла:

AxCalcControl.dll

Элемент управления Windows .NET

CalcControl.dll

Библиотека классов заместителей .NET

AxCalcControl.pdb

Отладочный файл

Добавление элемента управления в панель инструментов Visual Studio .NET

Импортировав элемент управления ActiveX, надо вернуться в окно Customize Toolbox, но на этот раз следует раскрыть вкладку .NET Framework Components. С помощью кнопки Browse... можно найти файл *AxCalcControl.dll*, созданный для элемента управления .NET Windows. Перенесите этот файл в окно Toolbox, как показано на рис. 22.9.

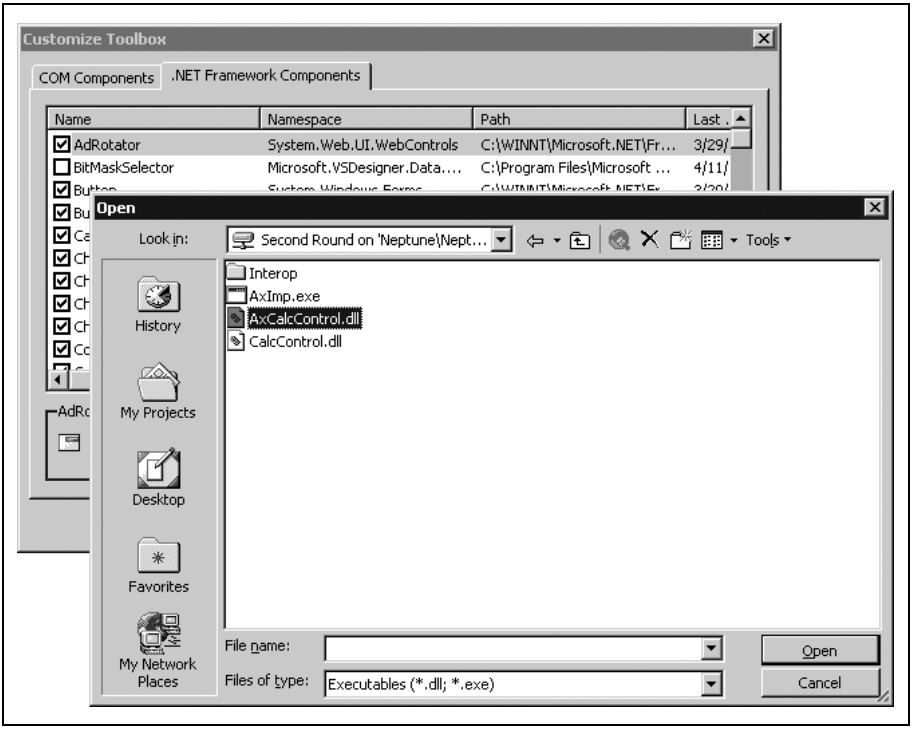

Рис. 22.9. *Поиск импортированного элемента управления*

После этих действий элемент управления появится в окне Toolbox (рис. 22.10). Имейте в виду, что он может появиться в самом низу окна, и тогда вы его не сразу заметите.

Теперь элемент управления можно перетаскивать на Windows-форму и пользоваться его функциями, как это было сделано в среде VB6.

Для каждой из четырех кнопок необходимо создать обработчик события. Эти обработчики будут делегировать свои обязанности элементу управления ActiveX, написанному на языке VB6 и импортированному в .NET.

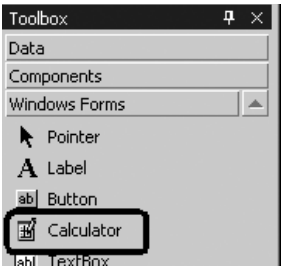

Рис. 22.10. *Калькулятор AxCalcControl, импортированный в панель инструментов*

Исходный текст обработчиков событий показан в примере 22.3.

Пример 22.3. Реализация обработчиков событий для тестирующей Windows-формы

```
private void btnAdd_Click(object sender, System.EventArgs e)
{
   double left = double.Parse(textBox1.Text);
   double right = double.Parse(textBox2.Text);
   label1.Text = axCalculator1.Add( ref left, ref right).ToString();
}

private void btnDivide_Click(object sender, System.EventArgs e)
{
   double left = double.Parse(textBox1.Text);
   double right = double.Parse(textBox2.Text);
   label1.Text = axCalculator1.Divide(ref left, ref right).ToString();
}

private void btnMultiply_Click(object sender, System.EventArgs e)
{
   double left = double.Parse(textBox1.Text);
   double right = double.Parse(textBox2.Text);
   label1.Text = axCalculator1.Multiply(ref left, ref right).ToString();
}

private void btnSubtract_Click(object sender, System.EventArgs e)
{
   double left = double.Parse(textBox1.Text);
   double right = double.Parse(textBox2.Text);
   label1.Text = axCalculator1.Subtract(ref left, ref right).ToString();
}
```

Каждый реализующий метод читает значения из текстовых полей, преобразует их в тип `double` с помощью статического метода `double.Parse()` и передает их методам калькулятора. Результаты вычислений преобразуются обратно в строковый тип и выводятся в поле статического текста, как показано на рис. 22.11.

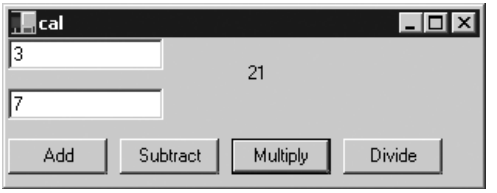

Рис. 22.11. *Выполнение импортированного элемента управления ActiveX в Windows-форме*

Импорт компонентов COM

Как оказалось, импорт элементов управления ActiveX не таит в себе ничего сложного. Однако многие COM-компоненты, предлагаемые на рынке программных продуктов, элементами ActiveX не являются и представляют собой стандартные файлы библиотек динамической компоновки (DLL). Чтобы узнать, как пользоваться ими на платформе .NET, вернемся в среду VB6 и создадим коммерческий объект COM, который будет работать точно так же, как элемент ActiveX из предыдущего раздела.

На первом шаге создадим новый проект ActiveX DLL. Именно таким образом VB6 создает стандартные библиотеки COM DLL. Назовем класс именем `ComCalc`, а проект – `ComCalculator`. Сохраните файл и проект. Методы, приведенные в примере 22.4, скопируем в текст программы.

Пример 22.4. Реализация методов для ComCalc

```
Public Function Add(left As Double, right As Double) As Double
    Add = left + right
End Function

Public Function Subtract(left As Double, right As Double) As Double
    Subtract = left - right
End Function

Public Function Multiply(left As Double, right As Double) As Double
    Multiply = left * right
End Function

Public Function Divide(left As Double, right As Double) As Double
    Divide = left / right
End Function
```

Скомпилируйте библиотеку DLL, выбрав в меню команду File → Make ComCalculator.dll. Чтобы ее протестировать, можно вернуться к предыдущей тестирующей программе и удалить с формы элемент управления Calculator. Добавьте новую библиотеку DLL, открыв окно References данного проекта и выделив ComCalculator, как показано на рис. 22.12.

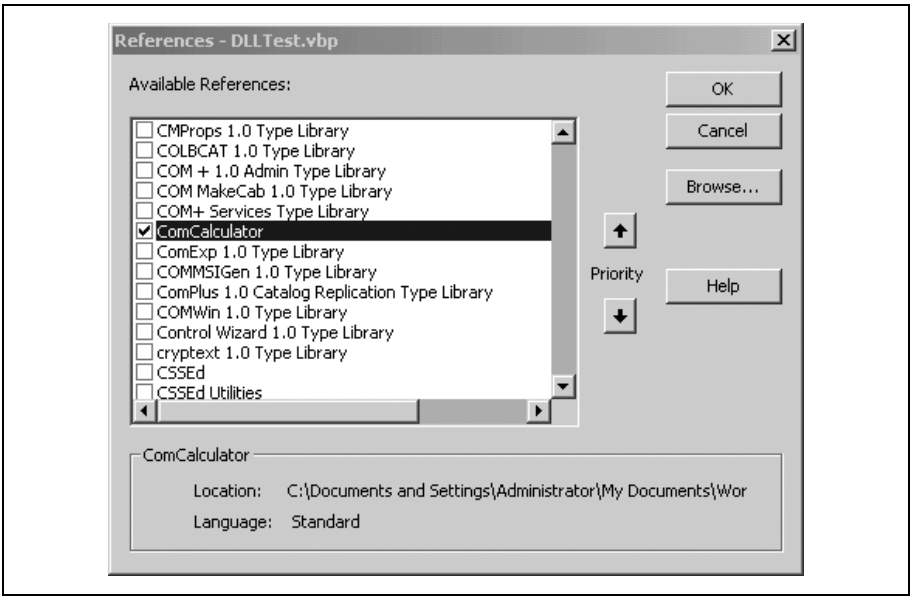

Рис. 22.12. *Добавление ссылки на библиотеку ComCalculator.dll*

Создание программы COMTestForm

Программа проверки компонента COM похожа на ту, что была рассмотрена в предыдущем примере. Впрочем, на сей раз создается объект `ComCalc` и вызываются его методы (см. пример 22.5).

Пример 22.5. Тестовая программа для ComCalc.dll

```
Private Sub btnAdd_Click()
    Dim theCalc As New ComCalc
    Label1.Caption = theCalc.Add(CDbl(Text1.Text), CDbl(Text2.Text))
End Sub

Private Sub btnDivide_Click()
    Dim theCalc As New ComCalc
    Label1.Caption = theCalc.Divide(CDbl(Text1.Text), CDbl(Text2.Text))
End Sub

Private Sub btnMultiply_Click()
    Dim theCalc As New ComCalc
    Label1.Caption = theCalc.Multiply(CDbl(Text1.Text), CDbl(Text2.Text))
End Sub

Private Sub btnSubtract_Click()
   Dim theCalc As New ComCalc
   Label1.Caption = theCalc.Subtract(CDbl(Text1.Text), CDbl(Text2.Text))
End Sub
```

Импорт COM DLL в среду .NET

Проверив DLL-библиотеку `ComCalc` в деле, ее можно импортировать на платформу .NET. Однако прежде всего следует сделать выбор между *ранним* и *поздним связыванием*. Когда клиент вызывает метод, находящийся на сервере, выполняется вычисление адреса этого метода в памяти. Такая процедура называется *связыванием*.

При *раннем связывании* вычисление адреса метода на сервере происходит на этапе компиляции проекта клиента и добавления метаданных к его модулю. В случае *позднего связывания* вычисление адреса выполняется лишь во время работы программы, когда COM-компонент ищет требуемый метод на сервере.

Ранее связывание имеет целый ряд достоинств. Самым важным из них является высокая производительность. При раннем связывании объекты вызываются гораздо быстрее, чем при позднем. Чтобы компилятор мог выполнить раннее связывание, он должен опросить библиотеку типов сервера, а для этого ее необходимо заранее импортировать на платформу .NET.

Импорт библиотеки типов

Файл COM DLL, созданный в среде VB6, имеет в своем составе библиотеку типов, однако формат библиотеки типов COM не позволяет использовать ее в приложении .NET. Чтобы решить эту проблему, следует сначала импортировать библиотеку типов COM в сборку. Как обычно, это можно сделать двумя способами. Либо программист регистрирует компонент, как показано в следующем разделе, и позволяет интегрированной среде Visual Studio .NET импортировать класс, либо он импортирует библиотеку типов вручную, с помощью специальной программы *TlbImp.exe*.

Эта утилита создаст сборку заместителя, содержащую манифест. Такая сборка заместителя называется RCW-сборкой (от англ. *Runtime Class Wrapper*, оболочка класса на этапе выполнения). Клиент платформы .NET будет использовать RCW-сборку для связывания методов COM-объектов, как показано в следующем разделе.

Ручной импорт

Скопируйте файл *ComCalculator.DLL* в среду .NET и зарегистрируйте его утилитой `Regsvr32`. Теперь можно импортировать COM-объект на платформу .NET с помощью утилиты *TlbImp.exe*. Ее синтаксис требует указания компонента COM и необязательного имени создаваемого файла (рис. 22.13).

Рис. 22.13. *Запуск утилиты TlbImp.exe*

Создание тестовой программы

Теперь настало время написать программу, тестирующую COM-объект. Назовем ее `COMDllTest`.

Если читатель решил не импортировать библиотеку вручную, ему следует сделать это с помощью Visual Studio .NET. В диалоговом окне Add Reference надо раскрыть вкладку `COM`, а на ней – выделить зарегистрированный объект COM, как показано на рис. 22.14.

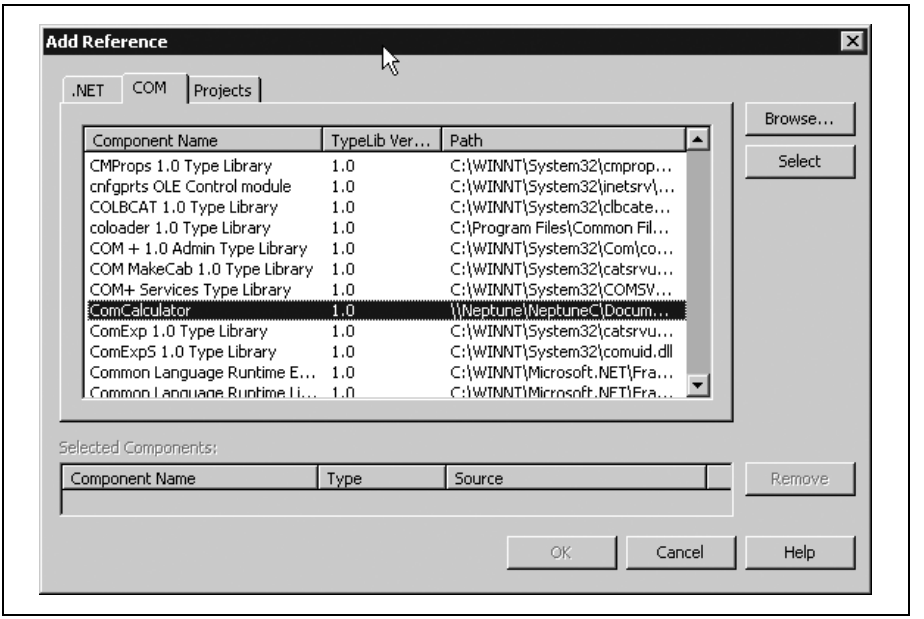

Рис. 22.14. *Добавление ссылки на ComCalculator*

Будет автоматически вызвана утилита *TlbImp*, которая скопирует RCW-сборку в каталог:

```
C:/Documents and Settings/Administrator/Application Data/Microsoft/
    VisualStudio/RCW
```

Здесь читатель должен быть максимально внимательным, поскольку создаваемая библиотека DLL будет иметь то же имя, что и библиотека COM DLL.

Если читатель все-таки решил вызвать утилиту *TlbImp.exe* вручную, надо добавить ссылку из вкладки Projects. Найдите каталог, в котором был создан файл *ComCalculatorDLLNET.dll* и добавьте эту библиотеку в список ссылок.

В любом случае теперь можно приступить к созданию пользовательского интерфейса. Как видно из рис. 22.15, он аналогичен интерфейсу, с помощью которого тестировался элемент управления ActiveX.

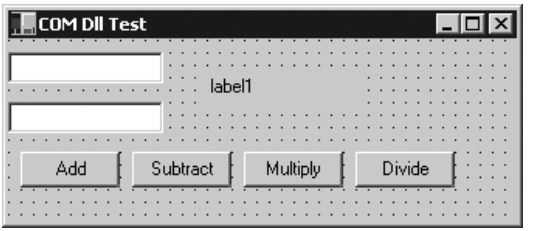

Рис. 22.15. *Форма для тестирования COM-объекта*

Теперь остается лишь создать обработчики событий для кнопок. Эти методы приведены в примере 22.6.

Пример 22.6. Реализация обработчиков событий для формы, тестирующей COM DLL из среды VB6

```
private void btnAdd_Click(
   object sender, System.EventArgs e)
{
   Double left, right, result;
   left = Double.Parse(textBox1.Text);
   right = Double.Parse(textBox2.Text);
   ComCalculator.ComCalc theCalc = new ComCalculator.ComCalc();
   result = theCalc.Add(ref left, ref right);
   label1.Text = result.ToString();
}

private void btnSubtract_Click(
   object sender, System.EventArgs e)
{
   Double left, right, result;
   left = Double.Parse(textBox1.Text);
   right = Double.Parse(textBox2.Text);
   ComCalculator.ComCalc theCalc = new ComCalculator.ComCalc();
   result = theCalc.Subtract(ref left, ref right);
   label1.Text = result.ToString();

}
```

```
private void btnMultiply_Click(
   object sender, System.EventArgs e)
{
   Double left, right, result;
   left = Double.Parse(textBox1.Text);
   right = Double.Parse(textBox2.Text);
   ComCalculator.ComCalc theCalc = new ComCalculator.ComCalc();
   result = theCalc.Multiply(ref left, ref right);
   label1.Text = result.ToString();
}

private void btnDivide_Click(
   object sender, System.EventArgs e)
{
   Double left, right, result;
   left = Double.Parse(textBox1.Text);
   right = Double.Parse(textBox2.Text);
   ComCalculator.ComCalc theCalc = new ComCalculator.ComCalc();
   result = theCalc.Divide(ref left, ref right);
   label1.Text = result.ToString();
}
```

На этот раз вместо того, чтобы сослаться на элемент ActiveX, расположенный на форме, необходимо создать объект `ComCalculator.ComCalc`. После этого COM-объект становится годным к применению, словно он был с самого начала создан в сборке .NET. Тестирующая программа работает как надо, что подтверждается рис. 22.16.

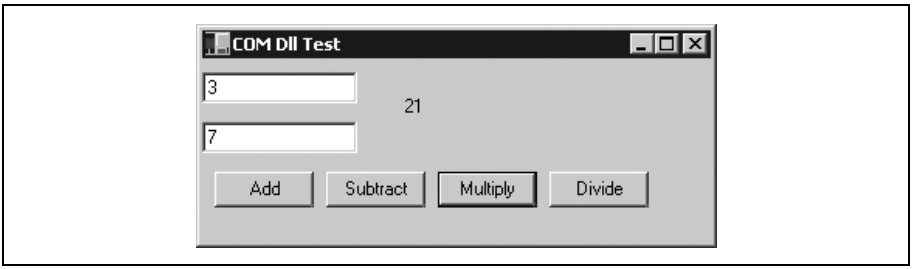

Рис. 22.16. *Тестовая программа в действии*

Применение позднего связывания и отражения

Если у разработчика нет библиотеки типов для COM-объекта, изготовленного третьей стороной (то есть не разработчиком и не его заказчиком), то он должен применить позднее связывание и отражение. В главе 18 было показано, как динамически вызывать методы в сборках на платформе .NET. Аналогичная процедура для объектов COM не имеет принципиальных отличий.

Чтобы убедиться в этом, возьмем программу из примера 22.6 и удалим ссылки на импортированную библиотеку. Обработчики событий долж-

ны быть переписаны. Создать экземпляр ComCalculator.Calc теперь невозможно, поэтому методы придется вызывать динамически.

Как и в главе 18, начнем с создания объекта Type, в котором будет храниться информация о типе ComCalc.

```
Type comCalcType;
comCalcType = Type.GetTypeFromProgID("ComCalculator.ComCalc");
```

Вызов метода GetTypeFromProgID() велит платформе .NET Framework открыть зарегистрированную библиотеку COM DLL и извлечь из нее информацию о типе указанного объекта. Этот метод представляет собой эквивалент метода GetType(), использованного в главе 18:

```
Type theMathType = Type.GetType("System.Math");
```

Далее можно продолжать так, словно вызывается метод класса, описанного в сборке .NET. Вначале получим экземпляр объекта ComCalc, вызвав метод CreateInstance():

```
object comCalcObject = Activator.CreateInstance(comCalcType);
```

Затем создадим массив для аргументов и вызовем нужный метод при помощи метода InvokeMember(). Методу InvokeMember() передаются пять аргументов: строка с именем вызываемого метода, флаг связывания, null в качестве связывающего класса, объект, возвращенный методом CreateInstance(), и массив входных аргументов:

```
object[] inputArguments = {left, right };
result = (Double) comCalcType.InvokeMember(
    "Subtract",                     // вызываемый метод
    BindingFlags.InvokeMethod,      // способ связывания
    null,                           // связывающий класс
    comCalcObject,                  // COM-объект
    inputArguments);                // аргументы метода
```

Результат, возвращаемый этим методом, приводится к типу Double и сохраняется в локальной переменной result. Значение этой переменной можно вывести в пользовательский интерфейс (рис. 22.17).

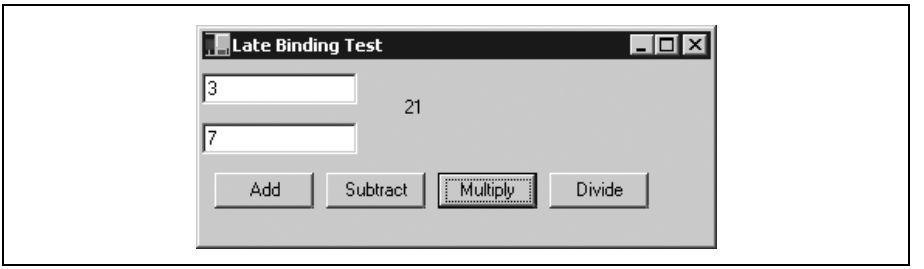

Рис. 22.17. *Тестирование позднего связывания*

Поскольку описанная процедура одинакова для всех обработчиков событий и различаются лишь имена вызываемых методов, общие операции можно выделить в закрытый вспомогательный метод по имени `Invoke()`, что и сделано в примере 22.7. Не забудьте добавить в исходный код оператор `using`, подключающий пространство имен `System.Reflection`.

Пример 22.7. Позднее связывание COM-объектов

```
private void btnAdd_Click(object sender, System.EventArgs e)
{
   Invoke("Add");
}

private void btnSubtract_Click(object sender, System.EventArgs e)
{
   Invoke("Subtract");
}

private void btnMultiply_Click(object sender, System.EventArgs e)
{
   Invoke("Multiply");
}

private void btnDivide_Click(object sender, System.EventArgs e)
{
   Invoke("Divide");
}

private void Invoke(string whichMethod)
{
   Double left, right, result;
   left = Double.Parse(textBox1.Text);
   right = Double.Parse(textBox2.Text);

   // создать объект Type для хранения информации о типе
   Type comCalcType;

   // массив для аргументов
   object[] inputArguments = {left, right };

   // получить информацию о типе от объекта COM
   comCalcType = Type.GetTypeFromProgID("ComCalculator.ComCalc");

   // создать объект
   object comCalcObject = Activator.CreateInstance(comCalcType);

   // вызвать метод динамически и привести результат к типу double
   result = (Double) comCalcType.InvokeMember(
      whichMethod,                // вызываемый метод
      BindingFlags.InvokeMethod,  // способ связывания
      null,                       // связывающий класс
      comCalcObject,              // COM-объект
      inputArguments);            // аргументы метода

   label1.Text = result.ToString();
}
```

Экспорт компонентов .NET

Класс .NET может быть экспортирован с целью использования его в существующих компонентах COM, хотя эта ситуация и не является типичной. Метаданные экспортируемого компонента регистрируются в системном реестре с помощью утилиты Regasm.

Вызывая эту утилиту, следует передать ей имя DLL-библиотеки, которая установлена в GAC (см. главу 17). Например:

```
Regasm myAssembly.dll
```

В результате метаданные компонента будут экспортированы в системный реестр. Давайте создадим новый проект C# DLL, в котором снова реализуем калькулятор на четыре действия (пример 22.8).

Пример 22.8. Калькулятор на четыре действия в составе DLL-библиотеки

```
using System;
using System.Reflection;

[assembly: AssemblyKeyFile("test.key")]
namespace Programming_CSharp
{
    public class Calculator
    {
        public Calculator(  )
        {

        }
        public Double Add (Double left, Double right)
        {
            return left + right;
        }
        public Double Subtract (Double left, Double right)
        {
            return left - right;
        }
        public Double Multiply (Double left, Double right)
        {
            return left * right;
        }
        public Double Divide (Double left, Double right)
        {
            return left / right;
        }
    }
}
```

Сохраним эту программу в файле *Calculator.cs* в рамках проекта ProgrammingCSharpDLL. Создайте строгое имя, скомпилируйте программу, добавьте ее в GAC и зарегистрируйте:

```
sn -k test.key
csc /t:library /out:ProgrammingCSharpDLL.cs Calculator.cs
```

```
gacutil /i ProgrammingCSharpDLL.dll
Regasm ProgrammingCSharpDLL.dll
```

Беглый просмотр системного реестра позволяет убедиться, что для DLL-библиотеки создан ключ PROGID (рис. 22.18).

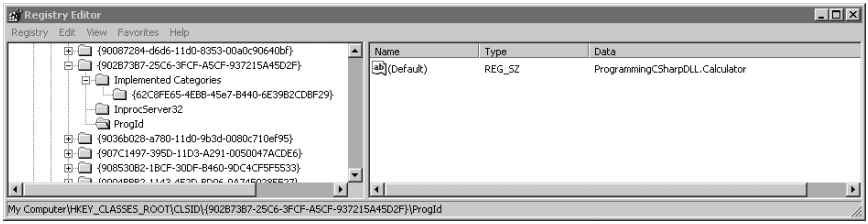

Рис. 22.18. *Реестр после регистрации DLL-библиотеки*

Теперь калькулятор можно вызывать как объект COM из программы на стандартном языке VBScript. Например, можно создать небольшой файл для сценария Windows, подобный тому, что приведен в примере 22.9.

Пример 22.9. Вызов COM-объекта Calculator из файла сценария Windows

```
dim calc
dim msg
dim result
set calc = CreateObject("Programming_CSharp.Calculator")
result = calc.Multiply(7,3)
msg = "7 * 3 =" & result & "."
Call MsgBox(msg)
```

Если выполнить этот сценарий, на экране появится диалоговое окно, позволяющее убедиться, что объект был создан и вызван (рис. 22.19).

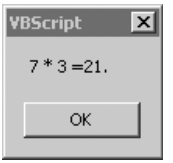

Рис. 22.19. *Позднее связывание с помощью модели COM*

Создание библиотеки типов

Если разработчик хочет, чтобы к его библиотеке .NET DLL было применено раннее связывание, он должен будет создать библиотеку типов. Это можно сделать с помощью утилиты экспорта библиотеки типов `TlbExp` (название которой представляет собой сокращение от Type Library Export), введя в командной строке следующую команду:

```
TlbExp ProgrammingCSharpDLL.dll /out:Calc.tlb
```

В результате будет создана библиотека типов, которую можно найти на диске и просмотреть с помощью утилиты OLE/COM Viewer инструментальной среды Visual Studio .NET, как показано на рис. 22.20.

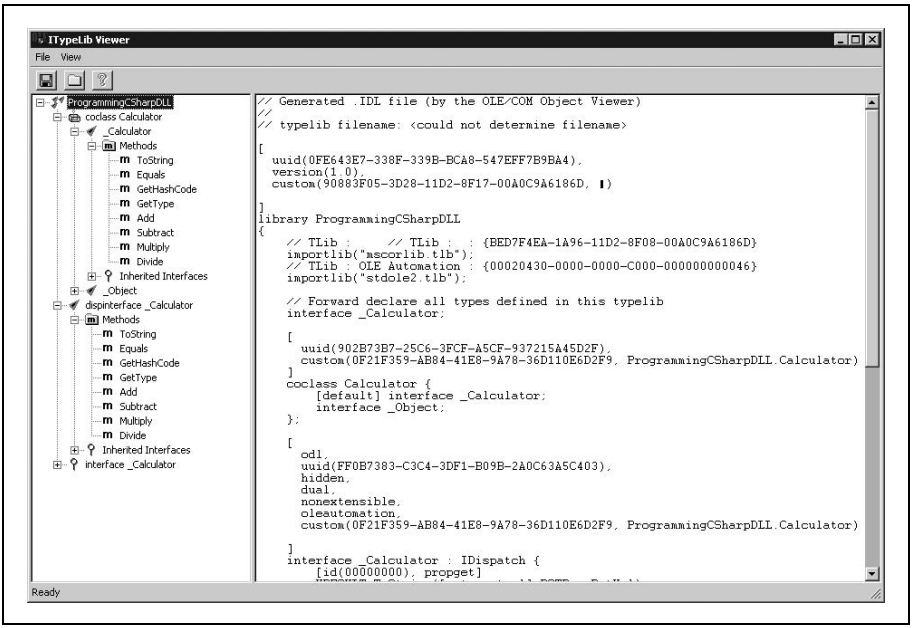

Рис. 22.20. *Просмотр содержимого библиотеки типов*

Имея эту библиотеку, разработчик может импортировать класс-калькулятор в любое COM-окружение.

Техника P/Invoke

Платформа .NET предоставляет программисту способ вызова неуправляемого кода из программы на языке C#. (Впрочем, делать это, вообще говоря, не рекомендуется.) Функциональная возможность `P/Invoke` (вызов *платформы*) первоначально задумывалась для предоставления доступа к Windows API, но пользоваться ею можно для вызова функций из любой библиотеки динамической компоновки.

Чтобы увидеть, как она работает, внесем изменения в пример 21.3 из главы 21. Читатель, конечно, помнит, как класс `Stream` использовался для переименования файлов методом `MoveTo()`:

```
file.MoveTo(fullName + ".bak");
```

То же самое можно сделать с помощью библиотеки Windows с именем *kernel32.dll*, точнее, ее метода `MoveFiles()`. Для этого следует объявить метод как `static extern` и поставить перед ним атрибут `[DllImport]`:

```
[DllImport("kernel32.dll", EntryPoint="MoveFile",
```

```
    ExactSpelling=false, CharSet=CharSet.Unicode, SetLastError=true)]
static extern bool MoveFile(string sourceFile, string destinationFile);
```

Класс DllImportAttribute служит в качестве индикатора того, что неуправляемый код будет вызван с помощью P/Invoke.

Он имеет следующие формальные параметры:

EntryPoint

Указывает имя точки входа в DLL-библиотеку (имя вызываемого метода).

ExactSpelling

Значение false показывает, что при поиске имени метода не учитывается регистр символов.

CharSet

Указывает способ передачи строковых аргументов метода.

SetLastError

Значение true позволяет вызвать метод GetLastError() и проверить, не возникла ли ошибка при вызове метода.

Остальная часть примера изменяться не должна, если не считать собственно вызова метода MoveFile(). Обратите внимание, что он вызывается как статический метод класса, в соответствии со своим объявлением:

```
Tester.MoveFile(file.FullName,file.FullName + ".bak");
```

Здесь методу передается первоначальное имя файла и новое имя. Файл перемещается, как при вызове метода file.MoveTo(). В рассматриваемом примере применение техники P/Invoke не дает никаких реальных преимуществ, но и не имеет существенных недостатков. Управляемый код остался, а результат не является объектно-ориентированным. Пользоваться функциональной возможностью P/Invoke имеет смысл лишь в тех случаях, когда разработчик просто вынужден вызвать метод, которому нет замены в рамках управляемого кода. В примере 22.10 приведен полный текст программы, применяющей P/Invoke для переноса файлов.

Пример 22.10. Применение техники P/Invoke для вызова метода Win32 API

```
namespace Programming_CSharp
{
   using System;
   using System.IO;
   using System.Runtime.InteropServices;

   class Tester
   {

      // объявить метод WinAPI, который будет вызван с помощью P/Invoke
      [DllImport("kernel32.dll", EntryPoint="MoveFile",
```

```
    ExactSpelling=false, CharSet=CharSet.Unicode,
    SetLastError=true)]
static extern bool MoveFile(
   string sourceFile, string destinationFile);

public static void Main()
{
   // создать объект и выполнить его
   Tester t = new Tester();
   string theDirectory = @"c:\test\media";
   DirectoryInfo dir = new DirectoryInfo(theDirectory);
   t.ExploreDirectory(dir);
}

// запустить, передав имя директория
private void ExploreDirectory(DirectoryInfo dir)
{

   // создать новый поддиректорий
   string newDirectory = "newTest";
   DirectoryInfo newSubDir =
      dir.CreateSubdirectory(newDirectory);

   // получить все файлы директория
   // и скопировать их в новый директорий
   FileInfo[] filesInDir = dir.GetFiles();
   foreach (FileInfo file in filesInDir)
   {
      string fullName = newSubDir.FullName +
         "\\" + file.Name;
      file.CopyTo(fullName);
      Console.WriteLine("{0} copied to newTest",
         file.FullName);
   }

   // получить коллекцию скопированных файлов
   filesInDir = newSubDir.GetFiles();

   // удалить одни файлы и переименовать другие
   int counter = 0;
   foreach (FileInfo file in filesInDir)
   {
      string fullName = file.FullName;

      if (counter++ %2 == 0)
      {
         // применить P/Invoke для Win API
         Tester.MoveFile(fullName, fullName + ".bak");

         Console.WriteLine("{0} renamed to {1}",
            fullName,file.FullName);
      }
      else
      {
```

```
                file.Delete();
                Console.WriteLine("{0} deleted.",
                   fullName);
             }
          }
          // удалить каталог
         newSubDir.Delete(true);
       }
    }
}
```

Вывод (отрывок):

```
c:\test\media\newTest\recycle.wav renamed to
    c:\test\media\newTest\recycle.wav
c:\test\media\newTest\ringin.wav renamed to
   c:\test\media\newTest\ringin.wav
```

Указатели

До сих пор в этой книге не было программ, использующих указатели в смысле C/C++. Эта тема затрагивается лишь здесь, в последнем разделе последней главы книги, несмотря на то что указатели являются центральным понятием в языках семейства C. Дело в том, что в языке C# указатели переведены в другую категорию и применяются в решении экзотических или нетривиальных задач программирования, как правило, при взаимодействии с моделью COM.

Язык C# поддерживает обычные операции над указателями, перечисленные в табл. 22.1.

Таблица 22.1. Операции над указателями в языке C#

Операция	Описание
&	Операция *адресации* возвращает указатель на адрес или значение
*	Операция *разыменования* возвращает значение по адресу, который определен указателем
->	Операция *доступа к элементу* используется для доступа к элементам типа

Пользоваться указателями, как правило, не рекомендуется, а необходимость в них возникает крайне редко. Применяя указатели, программист обязан пометить соответствующий фрагмент программы модификатором `unsafe`. Такой фрагмент считается небезопасным, поскольку указатели позволяют манипулировать памятью непосредственно – трюк, невозможный в обычной программе на языке C#. Из небезопасного кода программист может прямо обращаться к областям памяти, выполнять преобразования указателей в интегральные типы и обратно, получать адреса переменных и т. д. За это «удовольствие» он платит отсутствием сборки мусора и защиты от неинициализированных

переменных, висящими ссылками и отсутствием контроля границ. Можно сказать, что небезопасный код является островком C++ в безопасном приложении на языке C#.

В качестве демонстрации полезного применения указателей прочитаем файл и выведем его на экран с помощью двух методов API Win32: `CreateFile()` и `ReadFile()`. Метод `ReadFile()` в качестве второго формального параметра принимает указатель на буфер. Объявление двух импортируемых методов приведено в примере 22.11 и мало отличается от объявления, которое мы видели в предыдущем примере.

Пример 22.11. Объявление методов Win32 API для импорта их в программу C#

```
[DllImport("kernel32", SetLastError=true)]
static extern unsafe int CreateFile(
   string filename,
   uint desiredAccess,
   uint shareMode,
   uint attributes,
   uint creationDisposition,
   uint flagsAndAttributes,
   uint templateFile);

[DllImport("kernel32", SetLastError=true)]
static extern unsafe bool ReadFile(
   int hFile,
   void* lpBuffer,
   int nBytesToRead,
   int* nBytesRead,
   int overlapped);
```

Создадим новый класс по имени `APIFileReader`, конструктор которого будет вызывать метод `CreateFile()`. Конструктор принимает имя файла в качестве аргумента и передает его методу `CreateFile()`:

```
public APIFileReader(string filename)
{
   fileHandle = CreateFile(
      filename,       // имя файла
      GenericRead,    // способ доступа
      UseDefault,     // режим разделения
      UseDefault,     // атрибуты
      OpenExisting,   // способ создания
      UseDefault,     // флаги и атрибуты
      UseDefault);    // файл шаблона
}
```

Класс `APIFileReader` реализует только один дополнительный метод, `Read()`, вызывающий метод `ReadFile()`, передавая ему в качестве аргументов дескриптор файла, созданный конструктором класса, указатель на буфер, количество байт, которое следует прочитать, и ссылку на переменную, которая будет содержать количество прочитанных байт.

Из этих аргументов особый интерес представляет указатель на буфер. Чтобы выполнить вызов API, необходимо воспользоваться указателем.

Поскольку обращение к буферу происходит через указатель, буфер должен быть *зафиксирован* в памяти. Иными словами, должно быть запрещено перемещение буфера во время сборки мусора на платформе .NET Framework. Для этого в языке C# предусмотрено ключевое слово `fixed`. Оно позволяет получить указатель на область памяти, выделенную под буфер, и одновременно пометить этот объект как неперемещаемый сборщиком мусора.

Блок операторов, следующий за ключевым словом `fixed`, создает область, в пределах которой объекты зафиксированы в памяти. В конце блока объект снова становится перемещаемым. Это называется *декларативной фиксацией*:

```
public unsafe int Read(byte[] buffer, int index, int count)
{
   int bytesRead = 0;
   fixed (byte* bytePointer = buffer)
   {
      ReadFile(
        fileHandle,
        bytePointer +     index,
        count,
        &bytesRead, 0);
   }
   return bytesRead;
}
```

Обратите внимание, что метод должен быть помечен ключевым словом `unsafe`, что позволяет применять указатели и создает небезопасный контекст. Для компиляции такого метода необходимо указать параметр */unsafe*.

Тестовая программа создает объекты классов `APIFileReader` и `ASCIIEncoding`. Она передает имя файла конструктору класса `APIFileReader` и входит в цикл для заполнения буфера с помощью метода `Read()`, который создает API-вызов метода `ReadFile()`. Возвращенный байтовый массив преобразуется в строку с помощью метода `GetString()` объекта `ASCIIEncoding`. Далее строка передается методу `Console.Write()`, который выводит ее на экран. Полный исходный текст приведен в примере 22.12.

Пример 22.12. Применение указателей в программе на языке C#

```
using System;
using System.Runtime.InteropServices;
using System.Text;

class APIFileReader
{
   [DllImport("kernel32", SetLastError=true)]
```

```
   static extern unsafe int CreateFile(
      string filename,
      uint desiredAccess,
      uint shareMode,
      uint attributes,
      uint creationDisposition,
      uint flagsAndAttributes,
      uint templateFile);

   [DllImport("kernel32", SetLastError=true)]
   static extern unsafe bool ReadFile(
      int hFile,
      void* lpBuffer,
      int nBytesToRead,
      int* nBytesRead,
      int overlapped);

   // конструктор открывает существующий файл
   // и устанавливает его дескриптор
   public APIFileReader(string filename)
   {
      fileHandle = CreateFile(
         filename,      // имя файла
         GenericRead,   // способ доступа
         UseDefault,    // режим разделения
         UseDefault,    // атрибуты
         OpenExisting,  // способ создания
         UseDefault,    // флаги и атрибуты
         UseDefault);   // файл шаблона
   }

   public unsafe int Read(byte[] buffer, int index, int count)
   {
      int bytesRead = 0;
      fixed (byte* bytePointer = buffer)
      {
         ReadFile(
            fileHandle,             // дескриптор файла
            bytePointer + index,    // указатель на буфер
            count,                  // сколько байт прочитать
            &bytesRead,             // сколько байт прочитано
            0);                     // перекрывающийся ввод/вывод
      }
      return bytesRead;
   }

   const uint GenericRead = 0x80000000;
   const uint OpenExisting = 3;
   const uint UseDefault = 0;
   int fileHandle;
}

class Test
```

```
{
   public static void Main()
   {
      // создать объект класса APIFileReader,
      // и передать ему имя существующего файла
      APIFileReader fileReader =
        new APIFileReader("myTestFile.txt");

      // создать буфер и кодировщик ASCII
      const int BuffSize = 128;
      byte[] buffer = new byte[BuffSize];
      ASCIIEncoding asciiEncoder = new ASCIIEncoding();

      // прочитать файл в буфер и вывести на экран
      while (fileReader.Read(buffer, 0, BuffSize) != 0)
      {
         Console.Write("{0}", asciiEncoder.GetString(buffer));
      }
   }
}
```

Самый важный фрагмент этой программы выделен полужирным шрифтом. Здесь создается указатель на буфер, а буфер фиксируется в памяти ключевым словом `fixed`. Указателем пришлось воспользоваться, поскольку он необходим для вызова API. Впрочем, как показано в главе 21, то же самое можно было сделать и без API.

Приложение

Ключевые слова языка C#

`abstract`

Модификатор класса, показывающий, что для создания объекта этого класса следует создать объект его производного класса.

`ListGlossTerm`

Бинарная операция типа, осуществляющая приведение типа левого операнда к типу, указанному вторым операндом, и возвращающая `null`, не вызывая исключение, если преобразование закончится неуспешно.

`base`

Переменная, по смыслу равносильная `this`, с тем отличием, что она обращается к реализации члена базового класса.

`bool`

Логический тип данных, принимающий значение `true` или `false`.

`break`

Оператор выхода из цикла или оператора `switch`.

`byte`

Целочисленный тип данных без знака, имеющий размер в один байт.

`case`

Оператор выбора, определяющий конкретный пункт оператора `switch`.

`catch`

Часть оператора `try`, обрабатывающая исключения конкретного типа, указанного в аргументе оператора `catch`.

`char`

Символьный тип данных, определяющий двухбайтовую последовательность в кодировке Unicode.

`checked`

Оператор, устанавливающий проверку допустимости используемых значений в арифметических выражениях или блоках операторов.

`class`

Расширяемый ссылочный тип, объединяющий данные и функциональность в одной программной единице.

`const`

Модификатор объявления локальной переменной или поля, который показывает, что значение не меняется. Константа вычисляется на этапе компиляции и может иметь только базовый тип.

`continue`

Оператор перехода, который приводит к пропуску оставшихся операторов в блоке и возобновлению цикла со следующей итерации.

`decimal`

Десятичный тип данных размером 16 байт.

`default`

Метка в операторе `switch`, указывающая действие, которое следует предпринять, если ни один оператор `case` не соответствует выражению оператора `switch`.

`delegate`

Тип, определяющий сигнатуру метода, позволяющую объектам делегатов содержать и вызывать один или несколько методов, соответствующих сигнатуре.

`do`

Оператор цикла, многократно выполняющий блок операторов, пока выражение, указанное в конце, не примет значение `false`.

`double`

Тип данных с плавающей точкой; имеет размер 8 байт.

`else`

Условный оператор, определяющий действие, которое следует предпринять, если выражение в предшествующем операторе `if` имеет значение `false`.

`enum`

Размерный тип, определяющий группу именованных числовых констант.

`event`

Модификатор элемента класса для поля или свойства делегата, который означает, что доступны только такие методы делегата, как `+=` и `-=`.

`explicit`

Операция, определяющая явное преобразование типа.

`extern`

Модификатор метода, указывающий, что метод реализован неуправляемым кодом.

`false`

Логический литерал «ложь».

`finally`

Часть оператора `try`, которая всегда выполняется при передаче управления из блока `try`.

`fixed`

Оператор, фиксирующий ссылочный тип так, что сборщик мусора не будет перемещать его во время арифметических операций с указателями.

`float`

Тип данных с плавающей точкой; имеет размер 4 байта.

`for`

Оператор цикла, объединяющий в себе оператор инициализации, условие остановки и оператор итерации.

`foreach`

Оператор цикла, перебирающий элементы коллекций, реализующих интерфейс `IEnumerable`.

`get`

Название процедуры доступа, возвращающей значение свойства.

`goto`

Оператор перехода, передающий управление на метку в том же методе и той же области видимости, что и точка, в которой он стоит.

`if`

Условный оператор, выполняющий блок операторов, когда его выражение имеет значение `true`.

`implicit`

Операция, определяющая неявное преобразование типа.

`in`

Операция над типом и объектом `IEnumerable` в операторе `foreach`.

`int`

Тип целых чисел со знаком; имеет размер 4 байта.

`interface`

Контракт, указывающий элементы, которые класс или структура должны реализовать, для получения общих услуг для данного типа.

`internal`

Модификатор доступа, показывающий, что тип или его элемент доступен только типам в той же сборке.

`is`

Операция отношения, возвращающая `true`, если тип левого операнда либо соответствует типу, указанному правым операндом, либо является его потомком, либо реализует его.

`lock`

Оператор, устанавливающий блокировку на объект, имеющий ссылочный тип для обеспечения взаимодействия нескольких потоков.

`long`

Тип целых чисел со знаком; имеет размер 8 байт.

`namespace`

Связывает набор типов с общим именем.

`new`

Оператор, вызывающий конструктор типа, располагающий новый объект в куче, если он имеет ссылочный тип, или инициализирующий объект, если он имеет размерный тип. Это ключевое слово перегружается для сокрытия наследуемого элемента.

`null`

Литерал ссылочного типа, обозначающий отсутствие ссылки на какой-либо объект.

`object`

Тип, от которого произведены все остальные типы.

`operator`

Модификатор метода, перегружающего операцию.

`out`

Модификатор формального параметра, указывающий, что аргумент передается по ссылке, и его значение устанавливается вызываемым методом.

`override`

Модификатор метода, указывающий, что метод класса переопределяет виртуальный метод класса или интерфейса.

`params`

Модификатор формального параметра, указывающий, что последний параметр метода может принимать несколько аргументов одного типа.

`private`

Модификатор права доступа, показывающий, что только объемлющий тип имеет доступ к данному элементу.

`protected`

Модификатор права доступа, показывающий, что только объемлющий тип или типы, производные от него, имеют доступ к данному элементу.

`public`

Модификатор права доступа, показывающий, что тип или элемент типа доступен всем другим типам.

`readonly`

Модификатор поля, указывающий, что оно может получить значение только один раз, либо во время его объявления, либо при вызове конструктора типа.

`ref`

Модификатор формального параметра, указывающий, что аргумент передается по ссылке и получает значение до передачи его методу.

`return`

Оператор перехода, передающий управление из метода и указывающий возвращаемое значение, если метод не имеет тип `void`.

`sbyte`

Тип целых чисел со знаком; имеет размер 1 байт.

`sealed`

Модификатор класса, показывающий, что от этого класса нельзя производить другие.

`set`

Название процедуры доступа, устанавливающей значение свойства.

`short`

Тип целых чисел со знаком; имеет размер 2 байта.

`sizeof`

Операция, возвращающая размер структуры в байтах.

`stackalloc`

Операция, возвращающая указатель на заданное количество размерных типов, размещенных в стеке.

`static`

Модификатор члена типа, указывающий, что этот член относится к самому типу, а не к его экземпляру.

`string`

Встроенный ссылочный тип, представляющий неизменяемую последовательность символов в кодировке Unicode.

`struct`

Размерный тип, объединяющий данные и функциональность в одной программной единице.

`switch`

Оператор, предоставляющий ряд вариантов, выбираемых на основании значения базового типа.

`this`

Переменная, ссылающаяся на текущий объект класса или структуры.

`throw`

Оператор перехода, вызывающий исключение при возникновении ненормальной ситуации.

`true`

Логический литерал «истина».

`try`

Оператор, предоставляющий программисту способ обработать исключение или досрочно выйти из блока операторов.

`typeof`

Операция, возвращающая тип объекта в виде объекта `System.Type`.

`uint`

Тип целых чисел без знака размером 4 байта.

`ulong`

Тип целых чисел без знака размером 8 байт.

`unchecked`

Оператор, отменяющий проверку допустимости значения арифметического выражения.

`unsafe`

Модификатор метода или оператор, разрешающий применение арифметики указателей в пределах данного блока.

`ushort`

Тип целых чисел без знака размером 2 байта.

`using`

Указывает, что на типы в пределах данного пространства имен можно ссылаться, не квалифицируя их имена полностью. Оператор `using` определяет область видимости. В конце области видимости объект уничтожается.

`value`

Имя неявно существующей переменной, используемой в процедуре доступа `set()` свойства.

`virtual`

Модификатор метода класса, показывающий, что метод может быть переопределен производным классом.

`void`

Ключевое слово, используемое вместо имени типа для методов, не возвращающих никакого значения.

`volatile`

Указывает, что поле может быть модифицировано операционной системой или другим потоком.

`while`

Оператор цикла, многократно выполняющий блок операторов, пока выражение, указанное в начале, не примет значение `false`.

Алфавитный указатель

Специальные символы

- (вычитание), операция, 72
    приоритет, 79
-- (декремент), операция, 79
! (НЕ), операция, 77
    приоритет, 79
!= (не равно), операция отношения, 76, 79, 146
#, обозначение препроцессорных директив, 82
% (деление по модулю), операция, 68
    получение остатка, 72, 74
    приоритет, 79
%=, операция присваивания, 74, 79
&, операция адресации, 656
& (поразрядное И), операция, 79
&& (логическое И), операция, 77, 79
&=, операция присваивания, 79
' (кавычки), 47
() (круглые скобки), 268
* (умножение), операция, 72
    в качестве операции разыменования, 656
    приоритет операций, 78
*=, операция присваивания, 74, 79
+ (плюс)
    как операция сложения, 72
        приоритет, 78
    объединение делегатов, 317
++ (инкремент), операция, 79
+=, оператор, 326
    добавление делегата к множественному делегату, 317
+=, операция присваивания, 74, 79
. (доступ к элементу), операция, 33
    вывод текста на монитор, 31
/ (деление), операция, 72, 79
/* ... */ (комментарий в стиле C), 30
// (двойной слэш), обозначение комментариев, 30
    в XML-документе, 371
/=, операция присваивания, 74, 79
: (двоеточие), 267
    вызов конструктора базового класса, 125
; (точка с запятой)
    абстрактные классы, 134
    в конце оператора, 57
< (меньше), операция отношения
    приоритет, 79
<<>>, операции сдвигов, 79
<> (не равно), операция, 79
= (присваивание), операция, 55, 72
    приоритет операций, 78
-=, операция присваивания, 74, 79
=, операция присваивания, 79, 259
== (равно), операция отношения, 59, 76, 79, 146
    операции преобразования типов, 150
    строки, манипулирование, 259
@ (дословный строковый литерал), 253
    DirectoryInfo, объект, создание, 578
\ (обратный слэш), использование для esc-последовательностей, 47, 253
\' (апостроф), использование для esc-последовательностей, 47
^ (метасимвол), 268
^ (поразрядное исключающее ИЛИ), операция, 79
^=, операция присваивания, 79
{} (фигурные скобки), 69
    классы, определение, 87
    свойства и поведение, определение, 29
    элементы массива, инициализация, 198

| (поразрядное ИЛИ), операция, 79
| (вертикальная черта), 268
|=, операция присваивания, 79
|| (логическое ИЛИ), операция, 77, 79
~, операция, 79
> (больше), операция отношения, 59, 76, 146
-> (доступ к элементу), операция, 656
>= (больше или равно), операция отношения, 76
>=, операция отношения, 76, 146

A

\a (сигнал, esc-последовательность), 47
Abort(), метод, уничтожение потоков, 557
abstract, ключевое слово, 322, 661
Accept(), метод, 601
AcceptChanges(), метод, 409
AcceptSocket(), метод, 607
ActiveX, элементы управления, 634
Adapter(), метод, 226
Add(), метод, 146, 458, 468
    веб-службы, построение, 451
    индексирование, 214
    методы класса ArrayList, 226
    методы класса Hashtable, 244
AddRange(), метод, 227
ADO.NET, 24, 425
    XML, 424
    записи в базе данных, изменение, 407, 424
    использование, 390, 394
    объектные модели, 389, 390
    объекты, связанные с данными, 397
    управляемые поставщики, использование, 394, 397
All, цель атрибута, 484
AppDomain, класс, 527
Append(), метод, 265
append, аргумент, 593
AppendFormat(), метод, 265
AppendText(), метод, 581
Application Folder, папка Visual Studio .NET, 377
    адрес развертывания для приложений, 379
    специальные папки, добавление, 380
args, аргумент, 509
Array, класс, 208, 210
ArrayList, класс, 226, 357
as, оператор, 79, 175, 661
    отличие от оператора is, 176
ASP.NET
    C#, язык программирования, 447
    веб-формы
        жизненный цикл, 430
        элементы управления, добавление, 435
    отправляющие события, обработка, 445
.aspx, расширение файла, 432
    пользовательский интерфейс, 427
Assembly, цель атрибута, 484
Assembly.Load(), статический метод, 493
AssemblyInfo.cs, файл, 469
AssemblyLoad, событие, 528
AssemblyResolve, событие, 528
AssemblyResolver, загрузка сборок, 473
Attributes, свойство, 577, 582
AutoPostBack, свойство, 429
AxImp, утилита командной строки, 635, 640

B

base, ключевое слово, 661
BeginRead(), метод
    асинхронный ввод/вывод, 595
    выполнение двоичного чтения, 589
BeginWrite(), метод
    асинхронный ввод/вывод, 595
    выполнение двоичного чтения, 589
BinaryFormatter, класс, 622
BinaryReader, класс, 588
BinarySearch(), метод, 192, 227
BinaryWriter, класс, 588
binder, аргумент, 509
BindingFlags, параметр, 496
BizTalk 2000, 22
bool, тип, 45, 661
    значения по умолчанию, 93
Borland Delphi, 26
break, оператор, 58, 63, 65, 661
    continue, оператор, 70
BufferedStream, класс, 588
:button, объекты, 445
byte, тип, 45, 661

C

C#, язык программирования
.NET Framework, 21, 27
ASP.NET, 447
ключевые слова, 661
основы, 43, 85
C++, язык программирования, 21
стиль комментария, 30
стиль указателей, 656
Cab Project, опция Visual Studio .NET, 373
Calculator, класс, 542
camel, нотация, именование переменных, 54
Capacity(), метод, 265
Capacity, свойство, 226
CaptureCollection, 276, 278
case, оператор, 661
catch, операторы, 282, 283, 285, 661
корректирующие действия, 283
освобождение стека вызовов, 283, 285
специализированные, создание, 285
CDbl, функция VB6, 637
char, тип, 45, 47, 661
значения по умолчанию, 93
перечисления, 52
Chars(), метод, 265
Chars, поле, 254
CharSet, параметр, 654
checked, операция, 661
приоритет, 79
class, ключевое слово, определение типов как классов, 29, 661
Class, цель атрибута, 484
Clear(), метод
методы класса ArrayList, 227
методы класса Hashtable, 245
методы класса Stack, 240
методы класса SystemArray, 192
Clear
кнопка, обработка щелчка, 357
свойство, 238
Click, событие, 321
Clone(), метод
методы класса ArrayList, 227
методы класса Hashtable, 245
методы класса Queue, 238
методы класса Stack, 240
методы класса string, 254
close(), метод, реализация, 105
CLR (Common Language Runtime), общеязыковая среда выполнения, 23, 29, 476, 532
асинхронный ввод/вывод, 595
вычислительные потоки, 551
сериализация объектов, 621
CLS (Common Language Specification), общая языковая спецификация, 23, 44
Collections, пространство имен, 33
COM (Component Object Model), компонентная объектная модель
ActiveX, элементы управления, импортирование, 634
импортирование, 643
экспортирование, 651
commandString, параметр, 391
Common Language Runtime, см. CLR
Common Language Specification, см. CLS
Common Type System, см. CTS
Compare(), метод, 254
CompareOrdinal(), метод, 254
CompareTo(), метод, 225, 254
IComparable, реализация, 230
IComparer, реализация, 232
ComputeSum(), метод, 504
Concat(), метод, 254, 259
Configuration, пространство имен, 33
connectionString, параметр, 391
Console, объект
вывод текста на монитор, 31
операция принадлежности, 33
Console.ReadLine, метод, 70
Console.Write(), метод, 68, 147, 658
const, ключевое слово, 661
объявление констант, 51
Constructor, цель атрибута, 484
Contains(), метод
методы класса ArrayList, 227
методы класса Hashtable, 245
методы класса Queue, 238
методы класса Stack, 240
ContainsKey(), метод, 245
методы класса Hashtable, 245
continue, оператор, 58, 65, 70, 661
Copy(), метод, 192, 581
строки, манипулирование, 254
Copy, кнопка, обработка щелчка, 357, 361

CopyTo(), метод
FileInfo, класс, 582
ICollection, интерфейс, 225
методы класса Hashtable, 245
методы класса Queue, 238
методы класса Stack, 240
методы класса string, 255
файлы, модифицирование, 585
Cos(), метод, 497
вызов, 499
Count(), метод, 240
Count, свойство, 226, 237, 244
Create(), метод, 578, 581, 582
CreateChildControls(), метод, 430
CreateComInstanceFrom(), метод, создание объектов, 498
CreateDirectory(), метод, 576
CreateDomain(), метод, 527
CreateFile(), метод, 657
CreateInstance(), метод, 192, 531
CreateInstance(), метод, создание объектов, 498
CreateInstanceFrom(), метод, создание объектов, 498
CreateSubdirectory(), метод, 578
CreateText(), метод, 581
CreationTime, свойство, 577, 582
.cs-файлы, 432
CTS (Common Type System), общая система типов, 23
CurrentDomain, свойство, 527

D

Data, пространство имен, 33
DataAdapter, объект, 390
DataGrid, элемент управления, 397, 399
DataRelation, объект, 389
DataSet, класс, 389, 409
настройка, 399, 403
DataTable, объект, 389, 409
DBCommand, объект, 390
DBConnection, объект, 390
Debug, идентификатор, 82
decimal, тип, 45, 662
Decrement, метод, 565
default, ключевое слово, 662
#define, директива, 82
DefineDynamicAssembly(), метод, 527
delegate, ключевое слово, 662
Delegate, цель атрибута, 484
Delete(), метод, 576, 581
Delete, кнопка, обработка щелчка, 361, 371
Dequeue(), метод, 238
Dictionary, классы, разработка, 32
Directory
класс, 576, 588
свойство, 582
DirectoryInfo, класс, 588
Discovery, 449
Dispose(), метод, 105, 431
Div(), метод, 451
DLL (Dynamic Link Library), библиотека динамической компоновки, 27, 334, 432
COM-компоненты, импортирование, 643
многомодульные сборки, 466
сборки, 463
DllImportAttribute, класс, 654
do, оператор, 662
операторы цикла и итерации, 57
цикл while, 67
double, тип, 45, 662
DrawWindow(), метод, 41
Dynamic Link Library, см. DLL

E

#else, директива, 82
else, оператор, 58, 662
Emacs, компиляция программ, 35
Empty, поле, 254
#endif, директива, 82
EndRead(), метод, 597
EndsWith(), метод, 255, 260
Enqueue(), метод, 238
EnsureCapacity(), метод, 265
Enter(), метод, использование мониторов, 567
EntryPoint, параметр, 654
enum, тип, 45, 52, 662
значения по умолчанию, 93
Enum, цель атрибута, 484
Equals(), метод, 138, 146, 151, 255
методы класса string, 254
проверка равенства строк, 259
esc-последовательности, 47, 253
event, ключевое слово, 662
Event, цель атрибута, 484
EventArgs, класс, 322

ExactSpelling, параметр, 654
Excel, 333
Exchange 2000, 22
ExecuteAssembly(), метод, 527
EXE-файлы (выполняемые), 25, 27, 432
JIT-компиляция, 39
сборки, 463
многомодульные, 466
Exists(), метод, 576
Exists, свойство, 582
Exit(), метод, использование мониторов, 568
explicit, операция преобразования, 662
Extensible Markup Language (XML), расширяемый язык разметки
ADO.NET, 424
использование документирующих комментариев, 371, 372
Extension, свойство, 577, 582
extern, модификатор метода, 662

F

\f (перевод формата), esc-последовательность, 47
f(x), операция, приоритет, 79
false, значение, 45, 66, 662
операции отношения, 76
FCL (Framework Class Library), библиотека классов платформы, 23, 336, 428
пространства имен, 31
Field, цель атрибута, 484
FIFO (первым вошел, первым вышел), очередь, 237
File, класс, 588
FileInfo, класс, 588
FileStream, класс, 588
Finalize(), метод, 138
finally
блок, 280, 662
оператор, 287, 289
FindMembers(), метод, 496
findString(), метод, 216
fixed, оператор, 662
FixedSize(), метод, 226
float, тип, 45, 92, 662
FlowLayout, режим добавления элементов управления к веб-форме, 435
Flush(), метод, выполнение двоичного чтения, 589
font, объект, 106
for, оператор, 67, 69, 662
операторы цикла и итерации, 57
foreach, оператор, 70, 196, 210, 493, 662
IEnumerable, интерфейс, поддержка, 220
операторы цикла и итерации, 57
Form, объект, 335
Format(), метод, 254
Fraction, класс, 147, 150
Framework Class Library, см. FCL
FriendlyName, свойство, 527
FullName, свойство, 353, 577, 582
Func1(), метод, 283
Func2(), метод, 283

G

GAC (Global Assembly Cache), глобальный кэш сборок 476, 479
get(), метод, индексирование, 211
get, ключевое слово, 662
get, процедура доступа, 117
GetAttributes(), метод, 581
GetChanges(), метод, 409
GetCreationTime(), метод, 576, 581
GetCurrentDirectory(), метод, 576
GetCurrentThreadID(), метод, 527
GetData(), метод, 528
GetDirectories(), метод, 577, 578
GetDirectoryRoot(), метод, 577
GetEnumerator(), метод, 192, 220, 227, 238
методы класса Hashtable, 245
методы класса Stack, 240
GetFiles(), метод, 577, 578, 581
GetFileSystemInfos(), метод, 578
GetHashCode(), метод, 138, 245
GetLastAccessTime(), метод, 577, 581
GetLastWriteTime(), метод, 577, 581
GetLength(), метод, 192
GetLogicalDrives(), метод, 577
GetLowerBound(), метод, 192
GetMembers(), метод, 495
GetObject(), метод, создание объектов, 498
GetObjectData(), метод, 245
GetParent(), метод, 577
GetRange(), метод, 227
GetResponse(), метод, 619
GetString(), метод, 190, 658

GetType(), метод, 138, 494
серверы, создание, 542
GetUpperBound(), метод, 193
.gif-файлы, 463
Global Assembly Cache, см. GAC
goto, оператор, 58, 64, 66, 662
Graphical User Interfaces, см. GUI
GUI (Graphical User Interface), графический пользовательский интерфейс, 300
события, 320

H

Hejlsberg Anders, программист, 26
Hello World, программа, 28, 41
HelpLink, свойство, 290
HTML (Hypertext Markup Language), язык разметки гипертекста
Web Forms, 427, 431
WSDL-документ, просмотр, 455
веб-формы
базы данных, соединение, 440, 445
создание, 432, 434
элементы управления, добавление, 434, 436
применение XSLT- файлов, 372
HTTP-GET, протокол, 460
HTTP-POST, протокол, 460

I

Icalc, интерфейс, 544
ICloneable, интерфейс, 252
ICollection, интерфейс, 220, 225
IComparable, интерфейс, 252
реализация, 229, 232
IComparer, интерфейс, 220, 225
реализация, 232, 237
IConvertible, интерфейс, 252
IDE (Integrated Development Environment), интегрированная среда разработки, 35
IDictionary, интерфейс, 220, 246, 249
IDictionaryEnumerator, интерфейс, 220, 249
IDL (Interface Definition Language), язык определения интерфейса, 26, 485
IEnumerable, интерфейс, 220, 225, 252
#if, директива, 82
if, оператор, 58, 62, 175, 662
switch, оператор, 61
вложение, 60
Iformatter, интерфейс, 622
IList, интерфейс, 220
IL-файлы (см. MSIL-файлы), 25
implicit, оператор, 662
in, ключевое слово, операторы цикла и итерации, 57, 662
Increment, метод, 565
IndexOf(), метод, 192, 227, 260
Initialize(), метод, 193
InitializeComponent(), метод, 346
InnerException, свойство, 295
Insert(), метод, 227, 255, 265
InsertRange(), метод, 227
int, тип, 45, 662
get, метод доступа, 117
определение классов, 88, 92
прямоугольные массивы, 201
Integrated Development Environment, см. IDE
Interface Definition Language (IDL), язык определения интерфейсов 26, 485
interface, ключевое слово, 162
Interface, цель атрибута, 484
Interlocked, класс, 565
Intern(), метод, 254
internal protected, ключевое слово, 127
internal, модификатор прав доступа, 90, 663
Internet Explorer, броузер, 426
invokeAttr, аргумент, 509
InvokeMember(), метод, 504, 513
is, оператор, 173, 175
отличие от оператора as, 176
is, операция, 79, 663
IsFixedSize, свойство, 192, 226
IsInterned(), метод, 254
IsReadOnly, свойство, 192, 226
методы класса Hashtablee, 244
IsSynchronized, свойство
методы класса ArrayListt, 226
методы класса Hashtablee, 244
методы класса Queue, 237
методы класса Stack, 240
методы класса System.Array, 192
Item(), метод, 226, 244

J

Java, язык программирования, 21
JIT-компилятор, 25, 39
  использование, 39
Join(), метод, 254, 573

K

Keys, свойство, 243

L

language, атрибут, 433
LastAccessTime, свойство, 577, 582
LastIndexOf(), метод, 192, 227, 255
LastWriteTime, свойство, 577, 582
Length(), метод, 265
Length
  поле, 254
  свойство, 192, 582
LIFO (последним вошел, первым вышел), 240
ListChanged, событие, 321
Load(), метод, 528
Load, событие, 430
LoadPostData(), метод, 430
LoadViewState(), метод, 430
Locals, окно, 41
lock, оператор, 663
long, тип, 45, 46, 663
  операции преобразования типов, 147

M

Main(), метод, 29, 152
  IEnumerable, интерфейс, 224
  SingleCall, тип, 546
  static, ключевое слово, 35
  throw, оператор, 281
  асинхронный ввод/вывод, 596
  доступ к методам интерфейса, 172
  индексирование, 219
  классы, определение, 89
  конечные точки, 548
  консольные приложения, 31
  конструкторы, определение, 94
  освобождение стека вызовов, 283, 285
  параметры, передача, 108
  создание сервера, 542
  статические методы, вызов, 99
  структуры, создание, 158
Marshal(), метод, 548
MaxCapacity(), метод, 265
MemberFilter, параметр, 496
MemberTypes, параметр, 496
MemberwiseClone(), метод, 138
MemoryStream, класс, 588
Merge Module, опция Visual Studio .NET, 374, 377
Message, свойство, 289
Method, цель атрибута, 484
Microsoft Intermediate Language, см. MSIL
Module, цель атрибута, 484
Move(), метод, 577, 581
MoveFile(), метод, 654
MoveTo(), метод, 578, 582
  P/Invoke, 653
mscorlib, сборка, 466
MSIL (Microsoft Intermediate Language), промежуточный язык Microsoft
  JIT-компиляция, 39
  файлы, 25
Mult(), метод, 451
My Documents, папка, 380

N

\n (новая строка), esc-последовательность, 47
Name, свойство, 577, 582
namespace, ключевое слово, 663
.NET My Services, 448
.NET, платформа, 21
  ActiveX, элементы управления, импортирование, 634
  импортирование DLL, 645
  компоненты, экспортирование, 651
  программирование, 634
  элементы управления, импортирование, 638
Netscape Navigator, броузер, 426
NetworkStream, класс, 588
New Project, окно Visual Studio .NET, 339
new, ключевое слово, 322, 663
  конструктор базового класса, вызов, 125
  операция, 79
  создание версий, 132
  структуры, создание без new, 159

Next(), метод, 230
Notepad, текстовый редактор
Windows Form, создание, 339
веб-формы, выполнение, 427
компиляция программ, 35
элементы управления, создание, 336

O

Object, класс
хеш-таблицы, 245
Object, параметр, 496
object, тип, 663
OCX, стандарт, 634
OnDeserialization(), метод, 245, 627
OnLoad(), метод, 430
Open(), метод, 582
OpenRead(), метод, 581
открытие двоичных файлов, 589
OpenText(), метод, 582
OpenWrite(), метод, 581
открытие двоичных файлов, 589
operator, ключевое слово, 663
out, модификатор, 663
Outlook, приложение, 333
override, ключевое слово, 128, 322, 663
создание версий, 132

P

P/Invoke (платформный вызов), 653
P2P (peer to peer), равноправная связь, 600
PadLeft(), метод, 255
PadRight(), метод, 255
pageLayout, свойство, 435
Parameter, цель атрибута, 484
params, ключевое слово, 198, 663
Parent, свойство, 577
Pascal, нотация, именование переменных, 54
Passport, служба, 448
Peek(), метод, 238, 240
Perl 5
регулярные выражения языка, 268
PE-файлы, 463
Pop(), метод, 240
добавление и удаление элементов стека, 240
Pow(), метод, 451, 459
Primary Output, опция Visual Studio .NET, 375
private, модификатор прав доступа, 90, 663
ProcessExit, событие, 528
ProgCS, пространство имен, 468
ProgCSharp, параметр, 530
Project Output, меню Visual Studio .NET, 375, 377
Properties, окно Visual Studio .NET, 379
Property, цель атрибута, 484
protected, модификатор прав доступа, 90, 663
public, модификатор прав доступа, 87, 90, 127
Pulse(), метод, использование мониторов, 568
Push(), метод, 240
добавление и удаление элементов стека, 240

R

\r (возврат каретки), esc-последовательность, 47
RAD (Rapid Application Development), быстрая разработка приложений, 335, 427
RaisePostDataChangedEvent(), метод, 430
Rank, свойство, 192
Rapid Application Development, см. RAD
RCW (Runtime Class Wrapper), оболочка класса на этапе выполнения, 645
Read(), метод, 658
двоичные файлы, 589
реализация интерфейса, 163
переопределение, 177, 181
явная, 181
ReadFile(), метод, 657
ReadLine(), метод, работа с текстовыми файлами, 592
ReadOnly(), метод, 226
readonly, модификатор поля, 118, 663
ref, модификатор, 663
Refresh(), метод, 578
Regasm, средство экспортирования .NET-компонентов, 651

Regex, класс, 268
MatchCollection, коллекция, 271
группы, 272, 276
Regex, объект, 270
regexp, 268
#region, директива, 84
RegisterWellKnownServiceType(), метод, 547
Registry, окно Visual Studio .NET, внесение изменений, 381
RegularExpressions, пространство имен, 271
RejectChanges(), метод, 409
Remote Deploy Wizard, опция Visual Studio .NET, 374
RemotingConfiguration, класс, 542
Remove(), метод
методы класса ArrayList, 227
методы класса Hashtable, 245
методы класса string, 255
методы класса StringBuilder, 265
RemoveAt(), метод, 227
RemoveRange(), метод, 227
Repeat(), метод, 226
Replace(), метод, 265
Reset(), метод, 221
ResourceResolve, событие, 528
return, оператор, 58, 65, 663
ReturnValue, цель атрибута, 485
Reverse(), метод, 192, 227
ridLayout, режим добавления элементов управления к веб-форме, 435
Root, свойство, 578
Rows, коллекция, 389
Run(), метод, 323, 611
асинхронный ввод/вывод, 596
Runtime Class Wrapper, см. RCW

S

SaveViewState(), метод, 431
sbyte, тип, 45, 663
sealed, модификатор класса, 663
Serializable, атрибуты, 540
set(), метод, индексирование, 211
set
ключевое слово, 663
метод доступа, 118
SetAppDomainPolicy(), метод, 528
SetAttributes(), метод, 581
SetCreationTime(), метод, 576, 581
SetCurrentDirectory(), метод, 577
SetData(), метод, 528
SetLastAccessTime(), метод, 577, 581
SetLastError, параметр, 654
SetLastWriteTime(), метод, 577, 581
SetRange(), метод, 227
Setup Project, опция Visual Studio .NET, 374, 377
построение, 383
Setup Wizard, опция Visual Studio .NET, 374
SetValue(), метод, 193
Shape, класс, 533
short, тип, 45, 46, 664
Simple Object Access Protocol, см. SOAP
SingleCall, тип, 546
sizeof, операция, 664
приоритет, 79
Sleep(), открытый статический метод, 556
.sln-файлы, 432
SOAP (Simple Object Access Protocol), простой протокол доступа к объектам, 449, 532, 622
SoapFormatter, класс, 622
Solution Explorer, опция Visual Studio .NET, 376
Sort(), метод, 132, 192, 227
Copy, кнопка, обработка щелчка, 360
IComparable, реализация, 229
IComparer, реализация, 232
делегаты, 301
Split(), метод, 255, 270
SQL (Structured Query Language), язык структурированных запросов, 24, 438
реляционные базы данных, 384, 388
управляемый поставщик, 396
SQL Server 2000, СУБД, 22
Stack(), метод
добавление и удаление элементов стека, 240
stackalloc, операция, 664
приоритет, 79
Start(), метод, 601
StartRead(), метод, 607, 611
StartsWith(), метод, 255
static, ключевое слово, 35, 322, 664
Stream, класс, 588, 653

StreamReader, класс
  текстовые файлы, 592
StreamWriter, класс
  текстовые файлы, 592
string, тип, 664
StringBuilder, класс, 265, 267
StringReader, класс, 588
StringWriter, класс, 588
struct, ключевое слово, 45, 155, 664
Struct, цель атрибута, 485
Structured Query Language, см. SQL
Sub(), метод, 451
Subscribe(),метод, 325
Substring(), метод, 255
switch, оператор, 61, 664
Synchronized(), метод, 226, 237, 240, 244
SyncRoot, свойство, 192, 226, 238, 240, 244
System, пространство имен
  using, ключевое слово, 33
  операция принадлежности, использование, 33
System.Array, тип, 191
System.Drawing.Point, объект, 336
System.EnterpriseServices.Synchronization, атрибуты, 539
System.Exception, объект, 289
System.Reflection, пространство имен, 492
System.String, класс, 251
System.Text.RegularExpressions, пространство имен, 268
System.Threading, пространство имен, 324
System.Web, пространство имен, 427
System.Web.Services, пространство имен для серверной поддержки, 450
System.Web.UI, пространство имен, 427
SystemEventArgs, тип объекта, 337

T

\t (горизонтальная табуляция), esc-последовательность, 47
(T)x, операция, 79
TablesCollection, свойство, 389
target, объект, 509
TcpClient, клиент, 604
Tester, класс, 157
TextReader, класс, 588
TextWriter, класс, 588
this, ключевое слово, 98, 664
Thread, класс, 324, 553
ThreadAbortException, исключение, 558
throw, оператор, 58, 280, 664
TlbImp.exe, импортирование библиотеки типов, 645
ToArray(), метод, 227, 238, 240
ToCharArray(), метод, 255
ToLower(), метод, 255
Toolbox, панель инструментов Visual Studio .NET, 340
  базовая форма пользовательского интерфейса, 348
ToString(), метод, 138, 152, 253
  делегаты, 302, 304
  доступ к элементам массивы, 196
  структуры, создание, 157
ToUpper(), метод, 255
TreeView, элемент управления, 350, 354
  обработка событий, 354, 357
  обработка щелчка по кнопке Clear, 357
Trim(), метод, 255
TrimEnd(), метод, 255
TrimStart(), метод, 255
TrimToSize(), метод, 227
true, значение, 45, 66, 664
  операции отношения, 76
try
  блок, 283
  оператор, 664
Turbo Pascal, язык программирования, 26
Type, класс, 542
typeof(), метод, 492
typeof (получение типа), операция
  приоритет, 79, 664

U

uint, тип, 45, 664
ulong, тип, 46, 664
UML (Unified modeling Language), унифицированный язык моделирования, 121
unchecked (отключение проверки арифметики), операция
  приоритет, 79, 664

Unified Modeling Language, см. UML
Uniform Resource Identifier, см. URI
Unload(), метод, 527
unsafe, модификатор, 664
URI (Uniform Resource Identifier), унифицированный идентификатор ресурса, 544, 619
ushort, тип, 45, 46, 664
using System, оператор, 34
using, ключевое слово, 33, 106, 664
  идентификаторы, определение, 82

V

\v (вертикальная табуляция), esc-последовательность, 47
value, ключевое слово, 664
Values, свойство, 243
VB (Visual Basic), язык, 21, 335
View, меню Visual Studio .NET, 377, 380
ViewState, свойство, 430
virtual, ключевое слово, 322, 664
Visual Basic, см. VB
Visual Studio .NET
  ActiveX, элементы управления, импортирование, 635, 639
  Windows-приложения, создание, 339, 347
  XML-файлы, 371
  веб-формы, создание, 427
  отладчики, использование, 39, 41
  приложения, развертывание, 372, 383
  создание консольных приложений, 36
void, ключевое слово, 30, 664
volatile, ключевое слово, 664

W

Wait(), метод, использование мониторов, 568
Web Forms
  веб-приложения, программирование, 426, 447
  общая схема, 427, 431
  события, 428
Web Service Description Language, см. WSDL
Web Setup Project, опция Visual Studio .NET, 374
while, оператор, 66, 70
  операторы цикла и итерации, 57
Wiltamuth Scott, программист, 26
WindowClass, программа, 41
Windows.Forms, пространство имен, 335
Windows-приложения, построение, 333, 383
  TreeView, элемент, заполнение, 350, 354
  с помощью Visual Studio, 339
  с помощью Visual Studio .NET, 347
  с помощью Блокнота, 335
  формы, 335, 371
    ADO.NET, использование, 390, 394
    базовая форма пользовательского интерфейса, 348
Write(), метод
  двоичные оайлы, 589
  реализация интерфейса, 163
    переопределение, 177, 181
WriteLine(), метод, 33, 142, 190
  вывод текста на монитор, 31, 49
  работа с текстовыми файлами, 592
  структуры, создание, 157
  структуры, создание без операции new, 160
WSDL (Web Service Description Language), язык описания веб-службы, 449, 452
  документ, просмотр, 455
  файлы, создание полномочного класса, 457
while, оператор, 664

X

XE int, тип
  операции преобразования типов, 147
XE Location,структура, 155
XML (Extensible Markup Language), расширяемый язык разметки
  ADO.NET, 424
  SOAP, достоинства, 449
  документирующие комментарии, 371, 372
XSLT-файл, трансляция XML в HTML, 372

А

абстрактные классы, 120, 134, 137
завершенные, 137
ограничения, 136
отличие от интерфейсов, 176
адаптер данных, 390
адресация (&), операция, 656
апостроф ('), использование для esc-последовательностей, 47
аргументы
события, 428
метода, 91
асинхронный ввод/вывод, 594
атрибуты, 27, 483, 490
встроенные, 484
именование, 487
использование, 488, 490
конструирование, 487
объявление, 486
пользовательские, 484, 486, 490
применение, 485
цель, 484

Б

базовые классы, 123
конструкторы, вызов, 125
методы, вызов, 126
полиморфные методы, создание, 128
создание версий с помощью ключевых слов new/override, 132
базовые типы, 41, 44
выбор, 46
преобразование, 47
базы данных
веб-формы, соединение, 438
записи, изменение, 407, 424
реляционные, 384, 388
безусловное ветвление, операторы, 57
бесконечные циклы, 607, 611
библиотеки типов, 645, 652
бинарные файлы, 589
блок операторов, 60
блоки
catch, 282
finally, 288, 289
блокировки, синхронизация потоков, 566
Блокнот (Notepad), текстовый редактор Windows Form, создание, 335
больше (>), операция отношения, 59, 76, 146
приоритет, 79
броузеры
IP-адреса, 600
веб-службы, тестирование, 454
веб-формы, выполнение, 427
буферизованные потоки данных, 590

В

веб-потоки данных, 618
веб-приложения, 426, 447
веб-службы, 448, 460
построение, 450, 457
тестирование, 454
веб-формы
жизненный цикл, 429
отправляющие события, обработка, 445
связывание данных, 437, 445
создание, 431, 434
элементы управления, добавление, 434, 436
вертикальная табуляция, esc-последовательность (\v), 47
вертикальная черта (|), 268
ветвление, компиляция кода, 57
взаимная блокировка (тупиковая ситуация), 573
синхронизация потоков, 573
вложенные классы, 142
внешние
классы, 142
ключи, 385, 403
возврат каретки, esc-последовательность (\r), 47
временные данные, обработка, 627, 630
встроенные атрибуты, 484
выбор пункта меню, 300, 320
выражения, 55
вычитание (-), операция, 72
приоритет, 79

Г

горизонтальная табуляция, esc-последовательность (\t), 47
границы безопасности, 464

Д

данные
временные, обработка, 627, 630
чтение и запись, 588
двоеточие (:), 267
вызов конструктора базового класса, 125
двойные указатели на объекты, 531
декларативная
ссылочная целостность, 386
фиксация, 658
декларативные конструкции (см. атрибуты), 27
декремент, операция, 74
делегаты, 299, 320, 595
выбор методов на этапе выполнения, 301, 309
доступные только для чтения, 310
как свойства, 310
кнопки, создание, 337
множественное делегирование, 316, 320
размещение в массивах, 311, 316
события, 321, 329
статические, 310
деление (/), операция, 72, 79
по модулю (%), 68
получение остатка, 72, 74
приоритет, 79
деструкторы (C#), 104
структуры, 156
динамические строки, манипулирование, 265, 267
динамический вызов, 500
InvokeMember(), метод, 504
InvokeMember(), метод, 513
порождение отражения, 517, 524
с помощью интерфейсов, 513, 517
домены приложений, 527
контексты, 537
методы и свойства, 527
создание и использование, 529
упаковка, 531
дословные строковые литералы (@), 253
DirectoryInfo, объект, создание, 578
доступ к элементу
(-), операция, 656
(.), операция, 33
вывод текста на монитор, 31

Ж

жизненный цикл веб-формы, 429

З

завершенные классы, 137
закрытые
конструкторы, использование, 101
сборки, 474
записи в базе данных, 385, 407, 424
обновление, 412, 414
создание, 415, 424
удаление, 414
значения по умолчанию
базовые типы, 93
массивы, 193
значки, 300
зондирование, загрузка сборок объектом AssemblyResolver, 473

И

И (&&) логическое, операция, 77
идентификаторы, 54
определение, 82
отмена определения, 83
изолированная память, 630
ИЛИ (||) логическое, операция, 77
именование переменных, 54
именованные параметры, конструирование атрибутов, 487
имя метода, 509
индексаторы, 210, 219
операция, 246, 260
присваивание, 215
инициализаторы, 95, 97
инициализация в жизненном цикле веб-формы, 430
инкапсуляция, 87
инкремент, операция, 74
инструментальные средства Windows Forms, 339
Интернет, программирование веб-служб, 448, 460
интерфейсы, 26, 162, 190
коллекций, 220, 225
комбинирование, 167, 171
методы, доступ, 171, 177
методы, открытие, 184
отличие от абстрактных классов, 176

реализация, 163, 171
несколько интерфейсов, 166
переопределение, 177, 181
явная, 181, 190
серверные, 541
сокетов Беркли, 602
типы, 662
исключения, 298
catch, операторы, 282, 285
finally, оператор, 287, 289
вызов и перехват, 280, 289
объекты, 289, 292
повторный вызов, 295, 298
пользовательские, 293, 295

К

кавычки (\"), 47
каналы, 526
каталоги, 576, 578
DirectoryInfo, объект, создание, 578, 581
квадратные скобки, 193
классы, 24, 28, 35, 86, 119
абстрактные, 134, 137
базовые, 123
вызов методов, 126
вложенные, 141
завершенные, доступ, 185
изолированные, 120, 190
индексаторы, 210, 219
интерфейсы, реализация, 163, 171
методы, перегрузка, 112, 115
модификаторы прав доступа, 127
объекты, методы, 137
определение, 87, 92
отражение, 492
пользовательские исключения, 293
свойства, инкапсуляция данных, 115, 119
классы-посредники
создание, 457
клиентская поддержка, 450
клиенты
построение, 544
сетевые потоковые серверы, создание, 604
ключевые слова языка C#, 661
кнопка, объект, 337
кнопки, 300, 320, 347
Click, событие, 321
Copy, кнопка, обработка щелчка, 357, 361
Delete, кнопка, обработка щелчка, 361, 371
связывание данных, 437
коллекции
CaptureCollection, использование, 276, 278
MatchCollection, использование, 271
индексирование, 210
комментарии, 30
в XML-документе, 371, 372
в коде, 346
в стиле C (/* ... */), 30
в стиле C++, 30
компиляция, 25, 37
XML-файлы, 371
нормализация, 386
конечные точки, 542, 548
конкатенация (+), операция, 259
конкретные классы, 136
консольные приложения, 31
константы, 48, 55
перечисления, 52, 54
конструкторы, 92, 95
базовые классы, вызов, 125
закрытые, 101
определение, 93
перегрузка, 112, 115
статические, 100
структуры, вызов по умолчанию, 158
контекстно-связанные/свободные объекты, 538
контексты, 526, 537
контекстно-связанные и контекстно-свободные объекты, 538
упаковка, 539
контроль версий, 463, 482
конфликты, 245
копирование экрана, 621
копирующие конструкторы, 97
коэффициент загрузки, 247
круглые скобки (()), 268
куча, 46

Л

литералы, 267
null, ключевое слово, 663
константы, 50

логические операции, использование в условиях, 77
логические переменные, 45
операции отношения, 76
цикл while, 66
логическое И (&&), операция, 79
логическое ИЛИ (||), операция, 79

М

манифесты, 464, 466
модули, 466
маркер открытого ключа, 481
массивы, 191, 210
значения по умолчанию, 193
многомерные, 199, 206
объявление, 193
невыровненные, 203
преобразование, 206
размещение делегатов, 311, 316
списки, 225, 237
элементы, доступ, 194, 196
математические операции, 72, 74
меньше (<), операция отношения, 59, 76
метаданные, 464, 483
отражение, 491, 493
метасимволы, 267
методы, 29
Appdomain, класс, 527
базовый класс, вызов, 126
делегаты, 300
использование при выборе методов на этапе выполнения, 301, 309
интерфейса, 171, 177
открытие, 184
класс System.Array, 192
конструкторы, 92
отражение, 491
перегрузка, 112, 115
полиморфные, создание, 128
явная реализация, 182
многомодульные сборки, 466, 474
построение, 468, 474
тестирование, 472
множественное делегирование, 316, 320
модификаторы прав доступа, 90, 322
реализация интерфейса, 163
управление доступом, 126
модули, 463
манифесты, 466
многомодульные сборки, 466, 474
мониторы, синхронизация потоков, 567, 572

Н

наследование, 87, 120, 143
реализация, 123, 127
структуры, поддержка, 156
НЕ (!),операция, 77
приоритет, 79
не равно (!=), операция отношения, 76, 146
приоритет, 79
неизменяемые последовательности, 252
неотправляющие события, 429
неявное преобразование типов, 147
новая строка
esc-последовательность (\n), 47
пробельный символ, 55
ноль (0), использование для esc-последовательностей, 47
номера версий совместно используемых сборок, 477
нормализация, 386
нотация (camel/Pascal), именование переменных, 54

О

обобщение классов и объектов, 120
обработка
нескольких соединений, 606
полученных данных, 430
обработчики событий, 321, 428
обратный слэш (\), использование для esc-символов, 253
обратный слэш, использование для esc-последовательностей, 47
объектные модели, 389, 390
объектный граф, сериализация объектов, 621
объект-состояние, 595
объекты, 28, 35, 86, 119
ADO.NET, 389, 390
объекты, связанные с данными, 397
TreeNode, 350, 352

буферизованный поток данных, 590
двойные указатели, 531
делегаты, 301
десериализация, 624
исключения, 289, 292
контекстно-связанные и контекстно-свободные, 538
методы, 137
перегрузка, 112, 115
публикация/подписка, 321
свойства, инкапсуляция данных, 115, 119
связанные с данными, 397
серверные типы, 540
сериализация, 621
синхронизация потоков, 561, 572
события веб-форм, 428
создание, 92, 98
уничтожение, 103, 106
одиночные объекты, 540
одномодульные сборки, 466
одноразовые объекты, 540
операторы, 57, 71
catch, 282, 285
finally, 287, 289
throw, 280
безусловное ветвление, 57
перехода, 58
строка в операторе switch, 65
условное ветвление, 65
циклов, 65, 71
операции, 72, 80
operator, ключевое слово, 144
инкремент/декремент, 74
математические, 72, 74
над указателями в C#, 656
отношения, 76, 146
перегрузка, 144, 153
преобразование типов, 147
принадлежности (.)
вывод текста на монитор, 31
приоритет, 77
присваивания, 72, 79
распространенные ошибки, 62
создание, 146
определенное присваивание
передача параметров, 112
отдаление, 525
типы серверных объектов, 540
открытые
ключи, 478
маркеры, 481
свойства
AppDomain, класс, 527
Arraylist, 226
FileInfo, еласс, 582
Queue, 237
System.Array, 192
статические методы, 237
ArrayList, 226
Hashtablee, 244
System.Array, 192
потоки, приостановка, 556
стеки, 240
статические поля, для класса string, 254
отладчики, 38, 41
отправляющие события, 429
обработка, 445
отражение, 483, 490, 524
позднее связывание, 648
порождение (см. порождение отражения), 500
очереди, 237, 240
ошибки в программе, 279
ошибочные ситуации, 279

П

папки, 576
параметры, 91
именование, 145
определенное присваивание, возвращение значений, 112
передача, 106, 198
явная инициализация, возвращение значений, 109
первичные ключи, 385, 403
первым вошел, первым вышел (FIFO), очередь, 237
перевод формата, esc-последовательность (\f), 47
перегруженная операция присваивания (=), 259
передача файла, 320
переменные, 48, 55, 88
операции инкремента/декремента, 74
перечисления, 50, 54

перечислимый класс, 220
плюс (+), 259
как операция конкатенации, 259
подготовка к выводу в жизненном цикле веб-формы, 430
подписи (цифровые), 478
подписчики, 321
отделение от публикаторов, 329
подписывание сборки, 479
подстроки, поиск, 261, 263
позднее связывание, 491, 497, 645
отражение, 648
позиционные параметры, конструирование атрибутов, 487
полиморфизм, 87, 120, 143
методы, создание, 128, 131
создание типов, 127
полномочные классы
создание, 460
полномочный сервер, 450
получатели, 526
выстраивание с помощью полномочных классов, 532
поля, 87
пользовательские
атрибуты, 484, 486, 490
исключения, 293, 295
пользовательский интерфейс, 427
средства разработки, 31
формы, 348
управление в процессе настройки, 382
поразрядное
И (&),операция, 79
исключающее ИЛИ (^), операция, 79
порождение отражения, 500, 524
динамический вызов, 517, 524
порты, 600
последним вошел, первым вышел (LIFO), стек, 240
постфиксные операции, 75
потоки, 574
взаимная блокировка, 573
объединение, 556
приостановка, 556
синхронизация, 561, 572
Interlocked, класс, 565
блокировки, использование, 566
мониторы, использование, 567, 572
создание, 553, 555
состояние гонки, 573
уничтожение, 557, 561
потоки данных, 575
асинхронные сетевые файловые, 611
асинхронный ввод/вывод, 594
бинарные файлы, 589
буферизованные, 590
веб-потоки, 618
временные, обработка, 627, 630
изолированная память, 630
обработка нескольких соединений, 606
серверы, создание, 602
сетевой ввод/вывод, 600
сетевой клиент, создание, 604
чтение и запись данных, 588
потоковые серверы, создание, 602
преобразование
массивов, 206
типов, операции, 147
препроцессор, 82, 85
префиксные операции, 75
привязка данных, 437, 445
установка начальных значений свойств, 437
приложения, 426
для Windows, 333, 383
формы, создание, 335, 371
домены (см. домены приложений), 527
консольные, 31
развертывание, 372, 383
присваивание
индексаторы, 215
явные, 49
пробельные символы, 55, 69
программирование, ориентированное на события, 300
прозрачный заместитель, 532
производные классы, 123, 127
пространства имен, 31, 80, 452
процессы, 526
домены приложений, 527
прямоугольные массивы, 200, 203
публикаторы, 321
кнопки, создание, 337
отделение от подписчиков, 329

Р

равно (==), операция отношения, 59, 76, 146
операции преобразования типов, 150
приоритет, 79
строки, манипулирование, 259
разделители, 267
размерность массива, 200
разыменование (*), операция, 656
раннее связывание, 645
раскрытие типов, 491
распаковка типов, 139
реализует, отношение, 162
регулярные выражения, 267, 278
Regex, использование, 268, 270
реляционные базы данных, 384, 388

С

сборки, 27, 126, 463, 482
глобальный кэш, 479
закрытые, 474
многомодульные, 466, 474
совместно используемые, 474, 482
сборщик мусора, 46
исключения, 287
свойства, 27, 87
инкапсуляция данных, 115, 119
отражение, 491
связывание данных, 437
связывание, импортирование динамических библиотек COM DLL в .NET, 645
серверная поддержка, 450
серверы
интерфейсы, указание, 541
сетевые потоковые, создание, 602
создание, 541
типы объектов, 540
сериализация, 621, 623
объекты, создание потока данных, 576
форматизаторы, использование, 622
сетевой ввод/вывод, 600
сигнал, esc-последовательность (\a), 47
сигнатура метода, 112
символические константы, 50
символы Unicode, использование типа char, 47
синхронизация потоков, 561, 572
Interlocked, класс, 565
блокировки, использование, 566
синхронный ввод/вывод, 594
словари, 243, 250
коллекции ключей и значений, 248
сложение (+), операция, 72
слэш двойной (//), обозначение комментариев, 30
в XML-документе, 371, 372
события, 27, 320, 330
Copy, кнопка, обработка щелчка, 357, 361
Delete, кнопка, обработка щелчка, 361, 371
аргументы, 428
в веб-формах, 428
делегаты, 321, 329
обработчики, 321, 428
публикация/подписка, 321
совместно используемые сборки, 474, 482
версии, 477
построение, 479
чертовщина DLL, 476
соединения TCP/IP, 600
обработка, 606
сетевой потоковый клиент, создание, 604
сокеты, 600
обработка нескольких соединений, 606
состояние
веб-приложения, 429
гонки, синхронизация потоков, 573
сосуществование версий, 477
специализация классов и объектов, 120
список
базовых интерфейсов, 163
перечисления, 52
способы упаковки, 532
ссылочные типы, 95
значения по умолчанию, 93
статические
делегаты, 310
методы
вызов, 99
элементы
использование, 99, 103
конструкторы, применение, 100
поля, использование, 102

стеки, 46, 240, 243
catch, операторы, 283
вызовов, 290
столбцы, 385
строгие имена сборок, 476, 478
строки, 54, 251, 278
динамические, манипулирование, 265, 267
манипулирование, 254, 261
подстроки, поиск, 261, 263
создание, 252
строковый тип
в операторе switch, 65
структуры, 26, 154, 159
изолированные, 156
определение, 155, 157
создание, 157, 159
ступенчатые массивы, 206

Т

таблицы, 385
(см. также таблицы данных), 403
таблицы данных, объединение, 403, 407
табуляция (пробельный символ), 55
текстовый режим чтения файла, 589
текстовые
редакторы, компиляция программ, 35
файлы, 592
тернарная операция (?:), 79
технология шифрования строгих имен, 478
типы, 28, 35, 44, 48
библиотеки, 645, 652
значений, 44, 48
изолированные классы, доступ, 185, 190
стек и куча, использование, 46
структуры, 157
определяемые пользователем, 44
отражение, 491, 494, 496
получение информации о, 493
создание нового типа, 88
ссылочные, 44
указателей, 44
точка останова, компиляция и выполнение программы, 39
точка с запятой (;)
абстрактные классы, 134
в конце оператора, 57
тупиковая ситуация (взаимная блокировка), 573

У

указатели, 656
умножение (*), операция, 72
приоритет операций, 78
упаковка, 46, 525
домен приложения, 531
контексты, 539
с помощью заместителей, 532
способы, 532
типы, 139
управляемые поставщики
ADO, 397
OLE DB, 394
условия, использование в логических операциях, 77
условное ветвление, операторы, 57

Ф

файлы, 576, 581
CAB-, 375
адрес развертывания для приложений, 379
бинарные, 589
модифицирование, 584
поддержки, 427
специальные папки, добавление, 380
текстовые, 592
типы, регистрация, 382
фигурные скобки ({}), 69
классы, определение, 87
свойства и поведение, определение, 29
элементы массива, инициализация, 198
фиксирование буферов, 658
форматизаторы, 526, 532
использование для сериализации данных, 622
функции, 29
класса, 29
операторы ветвления, 57

Х

хеш-таблицы, 244

Ц

цели атрибутов, 484
цифровые подписи, 478

Ч

числовые типы, значения по умолчанию, 93
чувствительность к регистру, 34, 258

Э

экземпляры (см. объекты), 29
элементы, 87
  класса, 87
  массива, 194, 198
  сокрытие, 184
  управления, 88, 320
  ActiveX, импортирование, 634
  веб-формы, добавление, 434
  добавление к веб-форме, 436
  импортирование, 638
  создание с помощью Notepad, 336

Я

является, отношение
  наследование от абстрактного класса, 162
явная
  инициализация, передача параметров, 109
  реализация интерфейса, 181, 190
явное преобразование типов, 147, 153
явные присваивания, 49

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-038-3, название «Программирование на C#, 2-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.