Visual C# 2010
ПОЛНЫЙ КУРС

BEGINNING
Visual C# 2010

Karli Watson
Christian Nagel
Jacob Hammer Pedersen
Jon Reid
Morgan Skinner

Wiley Publishing, Inc.

Visual C# 2010
ПОЛНЫЙ КУРС

Карли Уотсон
Кристиан Нейгел
Якоб Хаммер Педерсен
Джон Рид
Морган Скиннер

Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75
У65
УДК 681.3.07

Компьютерное издательство "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *Я.П. Волковой, Ю.И. Корниенко, А.А. Моргунова, Н.А. Мухина*
Под редакцией *Ю.Н. Артеменко*

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

Уотсон, Карли, **Нейгел**, Кристиан, **Педерсен**, Якоб Хаммер, **Рид**, Джон Д., **Скиннер**, Морган.

У65 Visual C# 2010: полный курс. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2011. — 960 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1699-0 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Карли Уотсон, Кристиан Нейгел, Якоб Хаммер Педерсен,
Джон Д. Рид, Морган Скиннер

Visual C# 2010: полный курс

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

Подписано в печать 03.12.2010. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 77,4. Уч.-изд. л. 55,6.
Тираж 1500 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1699-0 (рус.)

ISBN 978-0-470-50226-6 (англ.)

Оглавление

Содержание

Об авторах

Карли Уотсон — консультант в Infusion Development (www.infusion.com), архитектор технологий в Boost.net (www.boost.net), а также внештатный специалист по информационным технологиям, автор и разработчик. По большей части увлекается .NET (особенно языком C#, а с недавнего времени — еще и WPF) и написал в этой области немало книг. Умеет передавать идеи так, чтобы они были понятны любому, кто жаждет знаний, и тратит массу времени на изучение любых новых технологий в поисках новых вещей, которым он еще может научить людей.

В те редкие периоды, когда Карли не занят ничем из того, о чем было упомянуто выше, он мечтает бороздить заснеженные склоны гор на сноуборде и, возможно, напечатать свой роман. В любом случае, его всегда можно узнать по яркой разноцветной одежде. Он доступен на www.twitter.com/karlequin. Карли выступил здесь автором глав 1–14, 21, 25 и 26.

Кристиан Нейгел — директор регионального представительства Microsoft и обладатель звания MVP, а также сотрудник компании Thinktecture (www.thinktecture.com) и владелец CN Innovation. Он занимается проектированием и разработкой программного обеспечения и проводит обучение и консультации по созданию решений на базе .NET. Начинал он свою карьеру с систем PDP 11 и VAX/VMS, разных языков и платформ. С 2000 г., когда технология .NET стала доступной для предварительного ознакомления, он начал работать с ней и создавать на ее базе различные решения. Получив благодаря этому глубокие знания в .NET, он написал множество книг и получил сертификат дипломированного инструктора и профессионального разработчика от Microsoft. Сегодня он регулярно выступает на международных конференциях, таких как TechEd и Tech Days, и запустил проект под названием INETA Europe для предоставления поддержки пользователям .NET. С ним можно связаться на веб-сайтах www.cninnovation.com и www.thinktecture.com, а также на странице www.twitter.com/christiannagel. В этой книге Кристиан написал главы 17–20.

Якоб Хаммер Педерсен — ведущий разработчик приложений в компании Elbek & Vejrup. Он начал программировать на языке BASIC. В начале 90-х годов прошлого столетия он занялся программированием для ПК, применяя сначала язык Pascal, но вскоре перешел на C++, который его интересует до сих пор его. В середине 90-х годов он снова сменил язык, которым на этот раз стал Visual Basic. А летом 2000 г. он открыл для себя язык C# и с тех пор не прекращает с удовольствием изучать его. Якоб в основном специализируется на платформах Microsoft, но помимо этого хорошо разбирается в разработке для MS Office, SQL Server, COM и Visual Basic.NET.

Являясь гражданином Дании, он проживает и работает в городе Орхус. В книге он выступил автором глав 15, 16 и 22.

Джон Рейд — руководитель отдела разработки программного обеспечения в компании Metrix LLC, которая является независимым поставщиком программного обеспечения для управления службами в средах Microsoft. Джон — соавтор множества книг по .NET, в том числе *Beginning Visual C# 2008*, *Beginning C# Databases: From Novice to Professional*, *Pro Visual Studio .NET*. Здесь он написал главы 23 и 24.

Морган Скиннер начал свою карьеру в юном возрасте на Sinclair ZX80 еще в школе, когда был поражен написанным учителем кодом, и с тех пор стал программировать на языке ассемблера. После этого он перепробовал массу других языков и платформ, в том числе VAX Macro Assembler, Pascal, Modula2, Smalltalk, X86, PowerBuilder, C/C++, VB и, конечно же, C#, которым увлечен и по сей день. Программированием для .NET он занимается с момента появления в 2000 г. выпуска PDC, который ему понравился настолько, что в 2001 г.

он присоединился к Microsoft. Сейчас работает главным консультантом по разработке приложений в Microsoft и помогает клиентам с языком C#. Морган написал в этой книге заключительную главу 27. С ним всегда можно связаться на веб-сайте `www.morganskinner.com`.

О техническом редакторе

Известный инженер и разработчик программного обеспечения на базе .NET в корпорации Intel Corporation с марта 2007 г., Дуг Холланд (Doug Holland) является членом группы Visual Computing Group и в настоящее время вместе с командой разработчиков использует развитые инструменты для тестирования наборов микросхем и драйверов. Является обладателем диплома магистра в сфере разработки программного обеспечения от Оксфордского университета, а также звания Microsoft MVP и Black Belt Developer от Intel. Помимо работы Дуг любит проводить время со своей женой и четырьмя детьми. Кроме того, он является служащим во вспомогательном подразделении гражданского воздушного патруля американских воздушных сил (Civil Air Patrol/U.S. Air Force Auxiliary).

Благодарности

Карли Уотсон. Спасибо всем в издательстве Wiley за поддержку и помощь в создании данного проекта, а также за понимание и гибкость, проявленные в работе с автором, у которого, похоже, постоянно не хватает времени писать. Особая благодарность моему редактору Эмми Салливан (Ami Sullivan) за привнесенную искру и придание данной книге настоящего лоска. Также спасибо моим друзьям, семье и коллегам по работе за понимание того, почему у меня не было особо времени общаться, а также, как всегда, Донне за ее поддержку и то, что она мирилась с моей работой допоздна.

Кристиан Нейгел. Спасибо двум моим девочкам — Анжеле и Стефани. Как хорошо, что вы у меня есть! Благодарю за вашу замечательную поддержку и всю ту любовь, которой вы меня окружали во время моего самого трудного периода жизни в 2009 г. Без вас я бы не справился. Стефани, хоть ты еще и не родилась тогда, ты была для меня самым главным стимулом. Люблю вас обеих!

Также выражаю благодарность моим соавторам и команде в издательстве Wrox/Wiley за выпуск столь замечательной книги.

Введение

C# является относительно новым языком, о котором мир впервые узнал, когда Microsoft в июле 2000 г. объявила о выходе первой версии .NET Framework. С тех пор он сильно вырос в плане популярности и стал чуть ли не самым предпочитаемым языком среди разработчиков Windows- и веб-приложений с использованием .NET. Отчасти привлекательным язык C# делает его понятный синтаксис, который происходит от синтаксиса C/C++, но позволяет более просто решать некоторые задачи. Несмотря на упрощение определенных вещей, язык C# все равно обладает той же мощью, что и C++, и потому на сегодняшний день нет причин, почему бы ни перейти на него. Он не сложен и замечательно подходит для изучения элементарных приемов программирования. Подобная простота изучения в сочетании с возможностями .NET Framework делают C# прекрасным выбором для начала программисткой карьеры.

В последней версии C# 4, поставляемой в составе .NET Framework 4, предлагаются как прежние успешные, так и новые функциональные возможности. В последней версии Visual Studio (Visual Studio 2010) и линейке продуктов для разработки Express (в том числе Visual C# 2010 Express) тоже появилось много настроек и улучшений для упрощения жизни разработчиков и значительного повышения их продуктивности.

Цель настоящей книги в том, чтобы продемонстрировать все аспекты программирования на C#. Материал начинается с концепций самого языка, способов написания Windows- и веб-приложений и работы с источниками данных и заканчивается описанием новых и более сложных приемов, а также возможностей Visual C# 2010 Express, Visual Web Developer 2010 Express и Visual Studio 2010 по разработке приложений.

Эта книга написана в дружественной, но при этом поучительной манере. Каждая глава вытекает из предыдущей, и приложен максимум усилий для безболезненного перехода к изучению более сложных приемов. Ни в одном месте технические термины не появляются ниоткуда; каждое понятие представляется и объясняется по мере необходимости. Употребление технического жаргона сведено к минимуму, но там, где без него не обойтись, в контексте обязательно приводятся соответствующие определения и пояснения.

Авторы книги являются экспертами в своей области и одинаково страстно увлекаются как C#, так и .NET Framework. Вряд ли удастся найти группу людей, более способных своими совместными знаниями обучить языку C#, начиная с простейших принципов и заканчивая более совершенными приемами. Помимо базовых сведений в книге также приводится масса полезных рекомендаций, советов, упражнений и полноценных примеров кода (доступных для загрузки на сайте издательства), к которым многие наверняка не раз будут обращаться на практике.

Авторы делятся здесь своими знаниями без утайки, и надеются, что они помогут читателю стать наилучшим в своем роде программистом. Удачи и всего самого наилучшего!

На кого рассчитана эта книга

Эта книга предназначена для любого, кто желает научиться программировать на языке C# с использованием .NET Framework. В первых главах рассказывается о самом языке и предполагается, что у читателя нет предыдущего опыта в сфере программирования. Тем, кто ранее программировал на других языках, большая часть материала в этих главах будет знакома. Многие аспекты синтаксиса в C# выглядят так же, как и в других языках, а многие структуры в нем (вроде структур для организации циклов и ветвления) присутствуют практически во всех языках программирования. Даже те, кто имеет опыт, все равно извлекут

пользу из прочтения этих глав, поскольку в них показано, как все эти приемы выглядят в языке C#.

Новичкам в программировании следует начинать с самого начала. Те, кто не знаком с .NET Framework, но умеет программировать, могут прочитать главу 1 и затем пропустить несколько следующих глав вплоть до начала применения изученных понятий C# на практике. Те, кто умеет программировать, но ранее не сталкивался с языками объектно-ориентированного программирования, могут переходить сразу к главе 8 и читать книгу далее.

Читатели, которым знаком язык C#, могут перейти сразу к изучению глав, посвященных недавним наработкам в .NET Framework и C#, в частности глав, посвященных коллекциям, обобщениям и улучшениям, появившимся в версии C# 4 (с 11 по 14); можно вообще пропустить всю первую часть книги и начать чтение с главы 15.

При написании глав преследовались две цели: сделать так, чтобы их можно было читать последовательно для получения полного курса обучения C#, и так, чтобы при необходимости в них можно было заглядывать выборочно для получения справочной информации.

Помимо основного материла в конце каждой главы (начиная с третьей) приводятся специально подобранные упражнения, решение которых поможет закрепить пройденный материал. Эти упражнения представляют собой как простые вопросы, предусматривающие выбор правильного ответа из набора вариантов или предоставление ответов в форме "да/нет", так и более сложные задачи, требующие внесения изменений в рассмотренные примеры приложений и написания собственного кода. Решения упражнений приведены в приложении.

Что нового в этом издании

Этому изданию настоящей книги было уделено повышенное внимание, поскольку его выпуск совпал с выходом версии C# 4 и .NET 4. Каждая глава была внимательно пересмотрена. Весь код был протестирован заново с использованием инструментов разработки самой последней версии, а все снимки экрана сделаны заново в среде Windows 7.

Ошибки, выявленные в предыдущих изданиях, были устранены, и многие пожелания читателей — учтены.

Ниже перечислены особенности настоящего издания.

- Приводятся дополнительные и улучшенные примеры кода для практического изучения.
- Рассматриваются все нововведения, которые появились в версии C# 4, начиная с простых языковых улучшений, таких как именованных и необязательных параметров, и заканчивая более совершенными приемами вроде применения вариантности в обобщенных типах.
- Упрощено описание сложных приемов для начинающих программистов, что позволит ознакомиться со всем необходимым без углубления в непонятные дебри.

Как организована эта книга

Эта книга состоит из шести частей, краткое описание которых приведено ниже.

- **Введение**. Здесь рассказывается о том, чему посвящена настоящая книга и на кого она рассчитана.
- **Язык C#**. В этой части освещены все аспекты языка C#, начиная с базовых концепций и заканчивая приемами объектно-ориентированного программирования.
- **Программирование Windows-приложений**. В этой части рассматривается разработка на C# приложений для Windows, а также их развертывание.

- **Программирование веб-приложений**. В этой части рассматривается процесс разработки и развертывания веб-приложений и веб-служб.
- **Доступ к данным**. В этой части показано, как использовать данные в своих приложениях, в том числе те, которые хранятся в файлах на жестком диске, в формате XML и внутри баз данных.
- **Дополнительные технологии**. В этой части описаны дополнительные способы применения C# и .NET Framework, в том числе технологии WPF, WCF и WF, которые впервые появились в .NET 3.0 и были улучшены в .NET 4.

В следующих разделах предлагается краткое описание глав в основных частях книги.

Язык C# (главы 1–14)

В **главе 1** показано, что собой представляет язык C# и как он вписывается в платформу .NET. Рассматриваются основные приемы программирования в этой среде и применение сред Visual C# 2010 Express (VCE) и Visual Studio 2010 (VS).

В **главе 2** начинается демонстрация написания приложений на языке C#. Здесь будет представлен синтаксис языка и показано его использование на примере нескольких простых приложений командной строки и Windows-приложений. Эти примеры помогут понять, насколько быстро и легко можно создавать приложения и вводить их в эксплуатацию. Также можно будет ознакомиться со средами разработки VCE и VS и основными окнами и инструментами, которые будут использоваться в остальных главах.

Далее рассматриваются базовые аспекты языка C#. В **главе 3** вы узнаете, что собой представляют переменные и как ими манипулировать. В **главе 4** будет показано, как улучшать структуру приложений за счет управления потоком выполнения (за счет добавления циклов и ветвлений). В **главе 5** рассматриваются более сложные типы переменных, такие как массивы. В **главе 6** обсуждаются вопросы инкапсуляции кода в функции, что упрощает выполнение повторяющихся операций и повышает читабельность кода.

К началу **главы 7** вы должны иметь все необходимые знания о языке, поэтому в ней рассматривается процесс отладки приложений. Вы научитесь выводить трассировочную информацию во время выполнения приложений и применять VS для перехвата ошибок и принятия решений по их устранению с помощью предлагаемой внутри этого продукта мощной отладочной среды.

В **главе 8** начинается знакомство с приемами объектно-ориентированного программирования (ООП), где первым делом объясняется значение этого понятия и что понимается под объектом. На первый взгляд ООП может казаться очень сложной технологией. Глава посвящена развенчанию этого мифа и объяснению преимуществ ООП.

В **главе 9** теоретические сведения применяются на практике: в ней показано, как использовать приемы ООП в своих приложениях C#. Сначала объясняется, как определять классы и интерфейсы. В **главе 10** показано как определять члены классов (поля, свойства и методы), а в конце главы начнется построение приложения карточной игры, которое будет разрабатываться на протяжении нескольких последующих глав.

После изучения приемов ООП в C# в **главе 11** демонстрируются наиболее типичные сценарии их использования, в том числе работа с коллекциями объектов и выполнение их сравнений и преобразований. **Глава 12** посвящена обобщениям, которые представляют собой полезный механизм в C#, впервые появились в .NET 2.0 и позволяют создавать очень гибкие классы. В **главе 13** продолжается рассмотрение языка C# и ООП и показываются дополнительные приемы, в частности — события, играющие очень важную роль при программировании, например, Windows-приложений. И, наконец, в **главе 14** описаны средства языка, которые появились в версиях C# 3.0 и C# 4.

Программирование Windows-приложений (главы 15–17)

Глава 15 начинается с объяснения основ программирования для Windows и применения для этого сред VCE и VS. Сначала описываются основные концепции, а в **главе 16** рассматривается использование в приложениях многочисленных элементов управления, поставляемых в .NET Framework. Будет показано, как в .NET графически строить Windows-приложения и компоновать сложные приложения с минимальными усилиями и временем.

В **главе 17** описаны приемы развертывания приложений, в том числе создание установочных программ.

Программирование веб-приложений (главы 18–20)

Эта часть построена по тому же принципу, что и часть, посвященная программированию Windows-приложений. В **главе 18** описаны элементы управления, входящие в состав простейших веб-приложений, и их совместное использование для решения различных задач с помощью ASP.NET. В главе также рассматриваются более сложные приемы, технология ASP.NET AJAX, разнообразные элементы управления, способы управления состоянием и существующие веб-стандарты.

В **главе 19** предлагается экскурс в замечательный мир веб-служб, которые позволяют программным образом получать доступ к данным и средствам через Интернет. Веб-службы позволяют снабжать веб- и Windows-приложения сложными данными и функциональностью не зависимым от платформы образом. В этой главе показано, как создавать и пользоваться веб-службами, а также описаны дополнительные средства, доступные для них в .NET, в том числе средства безопасности.

В **главе 20** рассматриваются способы развертывания веб-приложений и служб, в том числе публикация приложений с помощью нескольких щелчков.

Доступ к данным (главы 21–24)

В **главе 21** показано, как снабдить приложения возможностью сохранения и извлечения данных с диска, как в виде текстовых файлов, так и в виде более сложных представлений. Здесь также рассматривается сжатие данных, работа с унаследованными данными наподобие файлов со значениями, разделенными запятыми (CSV), и отслеживание изменений в файловой системе с принятием в их отношении тех или иных действий.

В **главе 22** можно будет ознакомиться с языком XML, который стал фактическим стандартом для обмена данными. В главе приводится список основных правил его использования и демонстрируются его преимущества.

В остальных главах настоящей части рассказывается о LINQ — языке запросов, который поставляется в составе всех последних версий .NET Framework. В **главе 26** показано, что собой представляет этот язык, а в **главе 24** — как его применять для доступа к информации, хранящейся в базе данных, и к другим данным.

Дополнительные технологии (главы 25–27)

В последней части книги рассматриваются новые технологии, которые появились в последних выпусках .NET Framework. В **главе 25** рассказывается о технологии WPF (Windows Presentation Foundation) и показано, к каким изменения она приведет в области разработки Windows- и веб-приложений. В **главе 26** речь идет о технологии WCF (Windows Communication Foundation), которая улучшает и расширяет технологию веб-служб и превращает ее в технологию для коммуникаций на уровне предприятия. **Глава 27** посвящена технологии WF (Windows Workflow Foundation), которая позволяет реализовать в приложениях механизм рабочих потоков, т.е. определять операции, выполняющиеся в определенном порядке в зависимости от внешних взаимодействий, что очень полезно для многих типов приложений.

Что необходимо для работы с этой книгой

Код и описания C# и .NET Framework, приводимые в настоящей книге, соответствуют версии .NET 4. Для их понимания кроме самой платформы .NET Framework больше ничего не требуется, но для проработки многих из предлагаемых здесь примеров требуется наличие какого-то средства разработки. В этой книге в качестве основного средства разработки применяется Visual C# 2010 Express, хотя в некоторых главах также встречается и Visual Web Developer 2010 Express. Кроме того, некоторые средства доступны только в полной версии Visual Studio 2010, на что обязательно указывается в соответствующих местах.

Соглашения

В книге используются следующие соглашения по представлению материала.

Во врезках с таким заголовком содержится важная информация, которая подлежит запоминанию и имеет непосредственное отношение к тому материалу, внутри которого находится.

Во врезках с таким заголовком содержатся примечания, советы, подсказки, приемы и прочие дополнения к текущему материалу.

В книге встречаются следующие стили.

- Новые термины и важные слова при первом упоминании выделяются *курсивом*.
- Клавиатурные комбинации приводятся в виде <Ctrl+A>.
- Имена файлов, URL-адреса и строки кода внутри текста выделяются моноширинным шрифтом: `persistence.properties`.
- Код выделяется двумя разными способами:
  - `Моноширинным шрифтом без полужирного начертания в большей части примеров кода.`
  - **`Моноширинным шрифтом с полужирным начертанием, если код является важным в текущем контексте.`**

Исходный код

Исходный код всех примеров, рассмотренных в книге, доступен для загрузки на вебсайте издательства по адресу `http://www.williamspublishing.com`.

От издательства

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

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

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

Наши координаты:

E-mail: `info@dialektika.com`

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

Информация для писем из:

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

Украины: 03150, Киев, а/я 152

ЧАСТЬ I

Язык C#

В ЭТОЙ ЧАСТИ...

1

Введение в C#

В ЭТОЙ ГЛАВЕ...

- Что такое .NET Framework и что она содержит
- Как работают приложения .NET
- Что собой представляет язык C# и как он связан с .NET Framework
- Средства для создания приложений .NET с помощью C#

Добро пожаловать в первую главу первой части этой книги. В этой части содержится весь базовый материал, который необходим для начала работы с языком C#. В настоящей главе дается краткий обзор языка C# и среды .NET Framework: что собой представляют эти технологии, для чего они применяются и как они связаны друг с другом.

Вначале приводится общее описание .NET Framework. В этой технологии имеется много понятий, разобраться в которых вначале не так-то просто. Из-за этого в настоящем разделе многие из новых понятий описываются очень кратко. Однако даже беглое рассмотрение базовых понятий важно для понимания того, как выполняется программирование на языке C#. Позже в данной книге многие из затронутых здесь тем будут рассматриваться снова и уже более детально.

После предоставления общих сведений далее в главе приводится базовое описание самого языка C#, его происхождения и схожих черт с языком C++. И в завершение вкратце рассказывается об основных средствах, которые используются в настоящей книге — Visual Studio 2010 (VS) и Visual C# 2010 Express (VCE).

Что такое .NET Framework

.NET Framework (теперь уже версии 4) — революционная платформа, созданная Microsoft для разработки приложений. Самым интересным в этом утверждении является его неоднозначность, для которой, однако, имеются вполне веские причины. Для начала обратите внимание: в нем не говорится, что эта платформа была создана для разработки приложений исключительно на базе операционной системы Windows. Выпущенная Microsoft версия .NET Framework рассчитана на работу под управлением Windows, однако существуют альтернативные версии для работы под другими операционными системами. Одним из примеров является версия Mono, которая распространяется с открытым исходным кодом (вместе с компилятором C#) и способна работать под управлением нескольких операционных систем, включая различные варианты Linux и Mac OS. Помимо этого, существует и версия Microsoft .NET Compact Framework, которая, по сути, представляет собой усеченный вариант полной версии .NET Framework и может применяться в устройствах класса карманных персональных компьютеров (КПК) и даже в некоторых смартфонах. Одним из главных стимулов к применению платформы .NET Framework является возможность использовать ее в качестве средства интеграции различных операционных систем.

Приведенное выше определение .NET Framework также не содержит никаких ограничений на типы возможных приложений — и таких ограничений действительно нет. Платформа .NET Framework позволяет создавать приложения Windows, веб-приложения, веб-службы и практически все остальные типы приложений, которые только можно себе представить.

Платформа .NET Framework спроектирована так, чтобы ее можно было использовать из любого языка, включая C# (тема настоящей книги), а также языков C++, Visual Basic, JScript и даже некоторых более старых языков, вроде COBOL. Для этого были выпущены специальные .NET-версии этих языков, и постоянно появляются новые. Все они не только имеют доступ к .NET Framework, но и могут взаимодействовать друг с другом. Это позволяет разработчикам на C# применять код, который был написан разработчиками на Visual Basic, и наоборот.

Все это открывает неимоверные возможности и делает .NET Framework очень привлекательным инструментом.

Что входит в состав .NET Framework

Платформа .NET Framework по большей части состоит из гигантской библиотеки кода, который можно использовать из клиентских языков (вроде C#) с помощью приемов объектно-ориентированного программирования (ООП). Эта библиотека поделена на различ-

ные модули, которые применяются в зависимости от требуемых результатов. Например, в одном модуле содержатся компоновочные блоки для приложений Windows, в другом — для программирования сетевого обмена, в третьем — для разработки веб-приложений. Некоторые модули разбиты на более конкретные подмодули: к примеру, модуль для разработки веб-приложений содержит подмодуль для создания веб-служб.

По замыслу разные операционные системы должны поддерживать некоторые или все эти модули, в зависимости от их характеристик. Например, КПК будет включать поддержку для всех основных функциональных возможностей .NET, но вряд ли потребует более специальные модули.

В одном из разделов библиотеки .NET Framework содержатся определения ряда базовых *типов*. Типы — это представление данных, и указание некоторых наиболее фундаментальных из них (например, "32-битное целое число со знаком") способствует совместимости между языками, использующими .NET Framework. Это носит название *общей системы типов* (Common Type System — CTS).

Помимо библиотеки, в состав .NET Framework входит и *общеязыковая исполняющая среда* (Common Language Runtime — CLR) .NET, которая отвечает за обслуживание процесса выполнения всех приложений, разработанных с помощью библиотеки .NET.

Написание приложений с помощью .NET Framework

Написание приложения с помощью .NET Framework означает написание кода с использованием одного из языков, поддерживающих .NET Framework, и кодовой библиотеки .NET. В настоящей книге для разработки приложений будут применяться средства VS и VCE. Первое представляет собой мощную интегрированную среду разработки, которая поддерживает код C# (а также управляемый и неуправляемый код на C++, Visual Basic и некоторых других языках), а VCE — усеченную (и бесплатную) версию VS, которая поддерживает только C#. Преимуществом этих сред является простота, с которой средства .NET могут интегрироваться в создаваемый код. Весь код будет полностью писаться на C#, но в нем везде будет использоваться .NET Framework и, где необходимо, другие дополнительные средства из VS и VCE.

Чтобы код на языке C# мог выполняться, он должен быть преобразован в код на языке, понятном целевой операционной системе. Такой код называется *машинным кодом* (native code), а сам процесс преобразования — *компиляцией*. Компиляция выполняется механизмом, который называется *компилятором*, и в .NET Framework состоит из двух этапов.

Язык CIL и JIT-компилятор

При компиляции кода, в котором используется библиотека .NET, он не преобразуется сразу же в код для конкретной операционной системы. Вместо этого он сначала преобразуется в код *CIL* (Common Intermediate Language — общий промежуточный язык). Этот код не является специфическим ни для какой-либо операционной системы, ни для языка C#. Другие языки .NET — например, Visual Basic .NET — на первом этапе тоже компилируются в код на этом языке. При разработке приложений на C# такой шаг компиляции выполняется с помощью VS или VCE.

Очевидно, что для запуска приложения потребуется выполнить еще кое-какую работу. За это отвечает так называемый *JIT-компилятор* (Just-in-Time compiler — оперативный компилятор), который компилирует CIL в машинный код, отвечающий требованиям конкретной операционной системы и архитектуры компьютера. Только после этого операционная система может запустить приложение. Аббревиатура *JIT* в названии компилятора означает, что CIL-код компилируется только при необходимости.

Раньше часто приходилось компилировать код в несколько приложений, каждое из которых предназначалось для конкретной операционной системы и архитектуры ЦП. Обычно так делали для оптимизации (например, чтобы код работал быстрее на микросхе-

мах AMD), но в некоторых случаях это было необходимостью (например, если нужно было, чтобы приложения могли работать как в средах Win9*x*, так и в средах WinNT/2000). Сейчас такой необходимости уже нет, поскольку JIT-компиляторы используют CIL-код, который не зависит ни от компьютера, ни от операционной системы, ни от процессора. Существуют несколько JIT-компиляторов, каждый из которых рассчитан на конкретную архитектуру, и для создания машинного кода применяется тот, который подходит в данном случае.

Вся прелесть этого механизма состоит в том, что он требует гораздо меньших усилий: разработчик может вовсе не думать о специфических деталях системы и сосредоточиться на более интересных фрагментах кода.

Иногда встречаются названия Microsoft Intermediate Language (MSIL) или просто IL. Первоначально CIL назывался MSIL, и многие разработчики до сих пор пользуются этим термином.

Сборки

При компиляции приложения создаваемый CIL-код сохраняется в *сборке* (assembly). В состав сборок входят исполняемые файлы приложений, которые имеют расширение `.exe` и могут запускаться прямо в среде Windows безо всяких других программ, и файлы библиотек, которые имеют расширение `.dll` и предназначены для использования другими приложениями.

Помимо CIL-кода, сборки также содержат *метаинформацию* (т.е. информацию о содержащихся в сборке данных, также называемую *метаданными*) и, не обязательно, файлы *ресурсов* (дополнительные данные, используемые в MSIL-коде, вроде звуковых файлов и изображений). Метаинформация позволяет сборкам быть полностью самостоятельными: для использования сборок не нужно никакой дополнительной информации, а это исключает ситуации вроде невозможности добавления требуемых данных в системный реестр и т.д., что часто было проблемой при разработках с помощью других платформ.

И тогда развертывание приложений обычно сводится просто к копированию файлов в каталог на удаленном компьютере. Никакой дополнительной информации на целевых системах не требуется, поэтому пользователь может просто запустить исполняемый файл из этого каталога и (если в системе установлена CLR-среда .NET) приступить к работе с приложением.

Конечно, вовсе необязательно хранить все необходимое для работы приложения в одном месте. Разработчик может писать и какой-нибудь код, решающий задачи, требуемые несколькими приложениями. В подобных ситуациях зачастую удобно поместить многократно используемый код в место, доступное всем приложениям. В .NET Framework таким местом является *глобальный кэш сборок* (Global Assembly Cache — GAC). Помещение в него кода осуществляется очень просто: нужно просто сохранить сборку, содержащую нужный код, в каталог, где находится этот кэш.

Управляемый код

Роль CLR-среды не заканчивается после компиляции кода в CIL-код и преобразования его JIT-компилятором в машинный код. Код, написанный с помощью .NET Framework, при выполнении (т.е. на этапе, который обычно называется *временем выполнения*) находится *под управлением* среды. Это означает, что CLR-среда обслуживает приложения: управляет памятью, обеспечивает безопасность, позволяет выполнять межъязыковую отладку и т.д. Приложения, которые выполняются не под управлением CLR-среды, называются *неуправляемыми* (unmanaged), и некоторые языки позволяют создавать такие приложения — например, для доступа к низкоуровневым функциям операционной системы. В языке C#, однако, разрешается писать только код, выполняющийся в управляемой среде, т.е. использовать

управляемые средства CLR и позволять среде .NET самостоятельно взаимодействовать с операционной системой.

Сборка мусора

Одной из наиболее важных возможностей управляемого кода является *сборка мусора* (garbage collection). Она гарантирует в .NET полное освобождение использованной приложением памяти, когда она уже не нужна. До появления .NET это было в основном заботой программистов, и несколько простых ошибок в коде могли привести к таинственному исчезновению крупных блоков памяти в результате их неправильного выделения. Обычно это приводило к постепенному замедлению работы компьютера и в конечном итоге к краху системы.

Механизм сборки мусора в .NET время от времени проверяет память компьютера и удаляет из нее все, что больше не нужно. Никакой периодичности для этой операции нет; она может выполняться тысячу раз в секунду, или каждые несколько секунд, или через любой другой промежуток времени, но она обязательно произойдет.

Программисты должны учитывать то, что данная операция автоматически выполняется через непредсказуемые промежутки времени. Код, требующий для выполнения большого количества памяти, должен осуществлять зачистку самостоятельно, а не ожидать, когда произойдет сборка мусора — это не так сложно, как кажется.

Что получается в итоге

Прежде чем продолжить, повторим рассмотренные выше шаги по созданию .NET-приложения.

1. Сначала пишется код приложения на совместимом с .NET языке, вроде C# (рис. 1.1).
2. Далее этот код компилируется в CIL-код, который сохраняется в сборке (рис. 1.2)
3. Перед выполнением этого кода (в результате либо запуска как исполняемого файла, либо запроса из другого кода) он вначале компилируется в машинный код с помощью JIT-компилятора (рис. 1.3).
4. После этого машинный код выполняется в контексте управляемой CLR-среды вместе со всеми другими запущенными приложениями или процессами, как показано на рис. 1.4.

Рис. 1.1. *Написание кода приложения на языке C#*

Рис. 1.2. *Компиляция кода приложения в CIL-код и сохранение его в сборке*

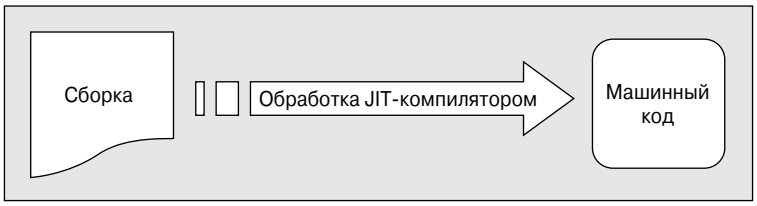


Рис. 1.3. *Компиляция CIL-кода в машинный код с помощью JIT-компилятора*

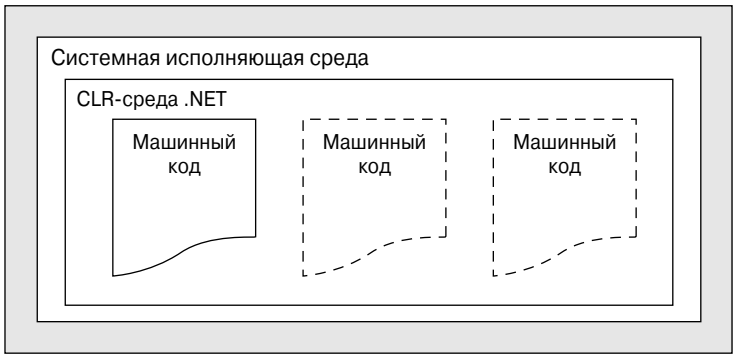


Рис. 1.4. *Выполнение машинного кода в контексте CLR*

Связывание

Обратите внимание на один дополнительный аспект в описанном процессе. Код C#, который на втором шаге компилируется в CIL-код, не обязательно должен содержаться в одном файле. Он может быть разбит на несколько файлов исходного кода, которые затем компилируются вместе в одну сборку. Этот чрезвычайно полезный процесс называется *связыванием* (linking). Дело в том, что гораздо проще работать с несколькими небольшими файлами, чем с одним огромным. Разработчик может выделять логически связанный код в отдельный файл, чтобы работать с ним отдельно, а затем практически забыть о нем. Кроме того, такой подход упрощает поиск конкретных фрагментов кода, когда они понадобятся, и позволяет командам разработчиков делить программирование на обозримые части, т.е. "сдавать" разработанные фрагменты кода, не боясь повредить уже готовые части кода или части, над которыми работают другие.

Что собой представляет язык C#

Как уже было сказано, язык C# является одним из языков, которые можно использовать для создания приложений, работающих в .NET CLR. Он был разработан Microsoft на базе языков C и C++ специально для работы с платформой .NET. В его состав вошли многие лучшие возможности из других языков, но без присущих им проблем.

Разрабатывать приложения на C# легче, чем на C++, потому что его синтаксис проще. Но все-таки C# достаточно мощный язык, и существует очень мало вещей, которые приходится делать на C++ из-за того, что их нельзя сделать на C#. Однако возможности C#, которые аналогичны более мощным возможностям языка C++, вроде прямых обращений к системной памяти, доступны только с помощью кода, помеченного как *небезопасный* (unsafe). Такие дополнительные приемы программирования потенциально опасны (поэтому такое название), поскольку они позволяют перезаписать критические для системы блоки памяти — возможно, с катастрофическими последствиями. По этой и некоторым другим причинам такие приемы в настоящей книге не рассматриваются.

Иногда код на C# немного более многословен по сравнению с C++. Объясняется это тем, что C# (в отличие от C++) представляет собой *безопасный к типам* (type-safe) язык. Попросту говоря, это означает, что после назначения данных какому-то типу их затем нельзя преобразовать в другой непохожий тип. Из-за этого преобразования между типами подчиняются строгим правилам, и для решения той же самой задачи на C# часто требуется писать более объемный код, чем на C++. Однако это дает определенные преимущества: код получается более надежным, отладка — более простой, а .NET всегда может определить тип любого фрагмента данных. Поэтому в C# невозможно, например, взять область памя-

ти размером 4 байта из данных длиной 10 байтов и интерпретировать это как X — хотя это не обязательно недостаток.

Язык C# — лишь один из языков, доступных для разработки .NET-приложений, но он, несомненно, лучший из них. Ведь это единственный язык, который с самого начала разрабатывался специально для .NET Framework, и поэтому он обязательно входит в состав версий .NET, переносимых на другие операционные системы. Чтобы языки вроде .NET-версии Visual Basic оставались максимально похожими на своих предшественников и все же совместимыми с CLR, некоторые средства из кодовой библиотеки .NET в них поддерживаются не полностью. В отличие от них, C# позволяет пользоваться всеми средствами этой библиотеки. В последней версии .NET в язык C# было добавлено несколько расширений (частично по просьбам разработчиков), которые еще более повышают его мощь.

Приложения, которые можно писать на C#

Как уже было сказано, в .NET Framework нет никаких ограничений на типы создаваемых приложений. C# использует эту среду и потому тоже не имеет никаких ограничений на типы создаваемых приложений. Ниже перечислены некоторые наиболее распространенные из них.

- **Windows-приложения**. Приложения, которые имеют знакомый пользователям Windows внешний вид, как, например, Microsoft Office. Для этого используется модуль Windows Forms (Формы Windows) из .NET Framework, представляющий собой библиотеку *элементов управления* (кнопок, панелей инструментов, меню и т.п.), из которых создается пользовательский интерфейс Windows (UI).
- **Веб-приложения**. Это веб-страницы, вроде тех, которые можно просматривать с помощью любого веб-браузера. В состав .NET Framework входит мощная система для динамической генерации веб-содержимого, которая обеспечивает персонализацию, безопасность и многое другое. Она называется ASP.NET (Active Server Pages .NET — активные серверные страницы .NET) и позволяет использовать C# для создания приложений ASP.NET с помощью модуля Web Forms. Кроме того, существует технология Silverlight, позволяющая создавать приложения, которые работают внутри браузера.
- **Веб-службы**. Замечательный способ для создания разнообразных распределенных приложений. Веб-службы позволяют обмениваться через Интернет практически любыми данными, используя одинаковый простой синтаксис, независимо от языка написания веб-службы и хост-системы. Для получения более широких возможностей можно также создавать службы Windows Communication Foundation (WCF).

Кроме того, приложениям любого из этих типов может потребоваться доступ к базам данным, который возможен с помощью либо средства .NET Framework под названием ADO.NET (Active Data Object .NET), либо технологии ADO.NET Entity Framework, либо LINQ (Language Integrated Query — язык интегрированных запросов) — средства самого языка C#. Могут быть задействованы и многие другие ресурсы: инструменты для создания сетевых компонентов, вывода графических объектов, выполнения сложных математических операций и т.д.

Язык C# в этой книге

В первой части этой книги описывается только синтаксис и применение языка C#, без особого внимания к .NET Framework. Такой подход был выбран потому, что работать с .NET Framework, не обладая базовым навыками программирования на C#, просто невозможно. Для еще большего упрощения пока не будет рассматриваться и объектно-ориентированное программирование (ООП). Изложение начинается с самых базовых понятий и не требует никаких знаний в области программирования.

После этого можно будет перейти к разработке более сложных (но и более полезных) приложений. Во второй части книги рассматривается программирование приложений Windows Forms, в третьей — программирование веб-приложений и веб-служб, в четвертой — доступ к данным (в базе данных, файловой системе или XML), а в пятой — некоторые другие интересные темы по .NET.

Visual Studio 2010

В настоящей книге для программирования на C#, начиная от простых приложений с интерфейсом командной строки и заканчивая более сложными проектами, используются средства разработки Visual Studio 2010 (VS) или Visual C# 2010 Express Edition (VCE). Средства разработки, а точнее, интегрированные среды разработки (Integrated Development Environment — IDE) вроде VS, не обязательны для создания приложений на C#, но они значительно упрощают его. При желании можно редактировать файлы исходного кода C# в простейшем текстовом редакторе, вроде Блокнота, и компилировать код в сборки с помощью компилятора командной строки, который входит в состав .NET Framework. Только зачем это делать, если есть мощные средства IDE-среды?

Ниже приведен краткий перечень некоторых возможностей Visual Studio, которые делают эту среду привлекательной для разработки .NET-приложений.

- VS автоматизирует шаги, требуемые для компиляции исходного кода, и в то же время предоставляет полный контроль на любыми используемыми опциями, если понадобится их переопределить.
- Текстовый редактор VS распознает синтаксис поддерживаемых языков (включая C#) и поэтому может интеллектуально обнаруживать ошибки и предлагать при вводе код, где это уместно. Данная возможность называется *IntelliSense*.
- В состав VS входят конструкторы для приложений Windows Forms, Web Forms и других типов, позволяющие добавлять элементы пользовательского интерфейса простым перетаскиванием.
- Многие типы проектов на C# могут создаваться на основе готового "шаблонного" кода. Вместо создания с нуля, разработчик часто может использовать различные кодовые заготовки, что сокращает время работы над проектом. Это особенно удобно в случае проекта типа Starter Kit (Начальный набор), который позволяет приступать к разработке на базе полностью функционального приложения. Некоторые такие проекты входят в состав VS, но еще больше их можно найти в Интернете.
- В VS имеется несколько мастеров, которые автоматизируют выполнение наиболее распространенных задач; многие из них автоматически добавляют нужный код в существующие файлы, позволяя не беспокоиться о синтаксисе (а иногда и напоминая его).
- В VS содержится много мощных средств для визуализации и навигации по элементам проекта, где бы они ни находились — в файлах исходного C#-кода или в файлах других ресурсов, вроде растровых изображений или звуковых файлов.
- В VS можно писать как просто приложения, так и целые проекты, облегчая передачу кода клиентам и его установку.
- VS позволяет использовать при разработке проектов более совершенные приемы отладки, например, возможность покомандного выполнения кода с одновременным отслеживанием состояния приложения.

Этот список можно продолжать долго, но он позволяет получить хотя бы общее представление.

Продукты Visual Studio 2010 Express

Помимо Visual Studio 2010, компания Microsoft поставляет несколько более простых средств разработки, которые называются продуктами Visual Studio 2010 Express (Visual Studio 2010 Express Products) и доступны бесплатно по адресу:

```
http://www.microsoft.com/express
```

Два из этих продуктов — Visual C# 2010 Express и Visual Web Developer 2010 Express — вместе позволяют создать на C# практически любое необходимое приложение. Оба они представляют собой усеченные версии VS с таким же внешним видом и поведением. В них имеются многие из тех же возможностей, что и в VS, и хотя некоторых заметных средств все-таки нет, они позволяют проработать материал настоящей книги.

Поэтому в этой книге везде, где можно, для разработки C#-приложений используется VCE и только если нужны особые возможности — полная версия VS. Конечно, при наличии VS можно не пользоваться продуктами Express.

Решения

При разработке приложений с помощью VS или VCE требуется создавать *решения* (solution). Решения в VS и VCE — это не просто приложения. Они содержат *проекты* (project): проекты Windows Forms, проекты Web Forms и т.д. Поскольку решения могут содержать несколько проектов, весь код можно группировать в одном месте, даже если впоследствии он будет скомпилирован в несколько сборок и будет храниться в разных местах на жестком диске.

Это очень удобно, поскольку позволяет работать и над совместно используемым кодом (который может размещаться в GAC), и над приложениями, в которых он применяется. Отладка кода существенно облегчается при наличии только одной среды разработки, т.к. это позволяет выполнять пошаговое выполнение инструкций в нескольких кодовых модулях.

Резюме

В этой главе было в общих чертах рассказано о среде .NET Framework и том, как она упрощает создание мощных и многофункциональных приложений. Вы узнали, что необходимо для превращения кода на языках вроде C# в рабочие приложения, и какие преимущества дает использование управляемого кода, запускаемого в среде .NET CLR.

Кроме того, вы узнали о том, что представляет собой язык C# и какое отношение он имеет к .NET Framework, а также об инструментах, которые можно применять для разработки приложений на C# — Visual Studio 2010 и Visual C# 2010 Express.

В следующей главе вы сможете выполнить простой C#-код, что позволит в дальнейшем сконцентрироваться на языке C#, не слишком заботясь о том, как работает IDE.

Что вы узнали в этой главе

Тема	Основные концепции
Базовые понятия .NET Framework	.NET Framework — последняя разработанная Microsoft платформа версии 4. Она содержит систему общих типов (CTS) и общеязыковую исполняющую среду (CLR). Приложения .NET Framework пишутся с помощью методологии объектно-ориентированного программирования (ООП) и обычно содержат управляемый код. Управление памятью, в том числе и сборка мусора, в управляемом коде выполняется с помощью исполняющей среды .NET.
Приложения .NET Framework	Приложения, написанные с помощью .NET Framework, вначале компилируются в CIL. Перед выполнением приложения JIT компилирует этот CIL в машинный код. Приложения компилируются, а различные части компонуются вместе в сборки, которые содержат CIL.
Основные сведения о C#	C# — один из языков, имеющихся в .NET Framework. Он создан на основе более ранних языков наподобие C++ и может применяться для написания любого количества приложений, в том числе веб-сайтов и приложений Windows.
Интегрированные среды разработки (IDE)	Visual Studio 2010 позволяет писать на C# любые виды .NET-приложений. Для создания .NET-приложений на C# можно также использовать бесплатный, хотя и менее мощный, набор продуктов Express (включая Visual C# Developer Express). Оба эти IDE-среды работают с решениями, которые могут состоять из нескольких проектов.

2

Написание программы на языке C#

В ЭТОЙ ГЛАВЕ...

- Основные сведения о Visual Studio 2010 и Visual C# 2010 Express Edition
- Написание простого консольного приложения
- Написание приложения Windows Forms

Теперь, когда вы уже знаете, что собой представляет язык C# и где его место в .NET Framework, можно написать какой-нибудь код. В этой книге везде будет использоваться либо Visual Studio 2010 (VS), либо Visual C# 2010 Express (VCE), поэтому вначале следует получить базовые сведения об этих средах разработки.

VS — огромный и сложный продукт, и неопытные пользователи могут вначале растеряться, но создание с его помощью простых приложений выполняется на удивление легко. Приступив к использованию VS в настоящей главе, вы увидите, что для того чтобы начать работать с кодом C#, вовсе не обязательно знать очень много. Потом будут продемонстрированы и более сложные операции, доступные в VS, но пока достаточно базовых навыков работы с компьютером.

VCE является гораздо более простым продуктом для начала работы, и потому на первых этапах в настоящей книге все примеры будут описываться в контексте этой IDE. Однако при желании вместо нее можно использовать VS, и все будет работать примерно так же. По этой причине в настоящей главе вы познакомитесь с обеими IDE-средами, начиная с VS.

После обзора IDE-сред вы создадите два простых приложения. Используемый в них код не важен; главное здесь убедиться, что все работает. Проработав процедуры создания приложений в этих первых примерах, вы быстро освоите их.

Первым будет создано простое *консольное приложение* (console application). Консольными называются такие приложения, в которых не применяется графическая оконная среда, что позволяет не заботиться о кнопках, меню, взаимодействии с курсором мыши и т.п. Вместо этого они запускаются в окне командной строки и взаимодействуют с пользователем гораздо более простым образом.

Вторым будет создано *приложение Windows Forms*. Внешний вид и поведение таких приложений хорошо знакомы пользователям Windows, и (что удивительно) их создание не требует намного больших усилий. Однако синтаксис необходимого для них кода выглядит сложнее, хотя на самом деле во многих случаях разработчику не нужно заботиться о многих деталях.

В следующих двух частях книги в примерах будут использоваться приложения обоих типов, хотя поначалу консольные приложения будут встречаться немного чаще. Дополнительная гибкость, которую обеспечивают приложения Windows, не нужна при изучении языка C#, а простота консольных приложений позволяет сконцентрироваться на изучении синтаксиса и не заботиться о внешнем виде приложения.

Среды разработки

В настоящем разделе описаны среды разработки VS и VCE, именно в таком порядке. Эти среды похожи между собой, и нужно прочитать оба раздела, независимо от того, какую из них вы собираетесь использовать.

Visual Studio 2010

При первой загрузке VS сразу же отображает целый ряд окон, большинство из которых пусто, а также набор пунктов меню и значков в панели инструментов. При чтении книги вы будете использовать большинство из них, поэтому вскоре они станут выглядеть вполне знакомо.

При первом запуске VS на экране появится перечень параметров, предназначенный для пользователей, у которых имеется опыт работы с предыдущими версиями данной среды. Выбираемые здесь настройки влияют на целый ряд аспектов, например, на расположение окон, на способ запуска консольных окон и т.д. Поэтому выберите в этом перечне опцию Visual C# Development Settings (Параметры разработки Visual C#), иначе позже может оказаться, что все работает несколько не так, как описывается в книге. Перечень доступных опций зависит от того параметров, выбранных при установке VS, но если во время инсталляции была выбрана установка C#, эта опция будет доступна обязательно.

Если же это не первый запуск VS, но в первый раз была выбрана другая опция, не волнуйтесь. Чтобы сбросить настройки и выбрать опцию Visual C# Development Settings, необходимо просто импортировать их. Для этого выберите в меню Tools (Сервис) пункт Import and Export Settings (Параметры импорта и экспорта), и затем выберите вариант Reset All Settings (Сбросить все настройки), как показано на рис. 2.1.

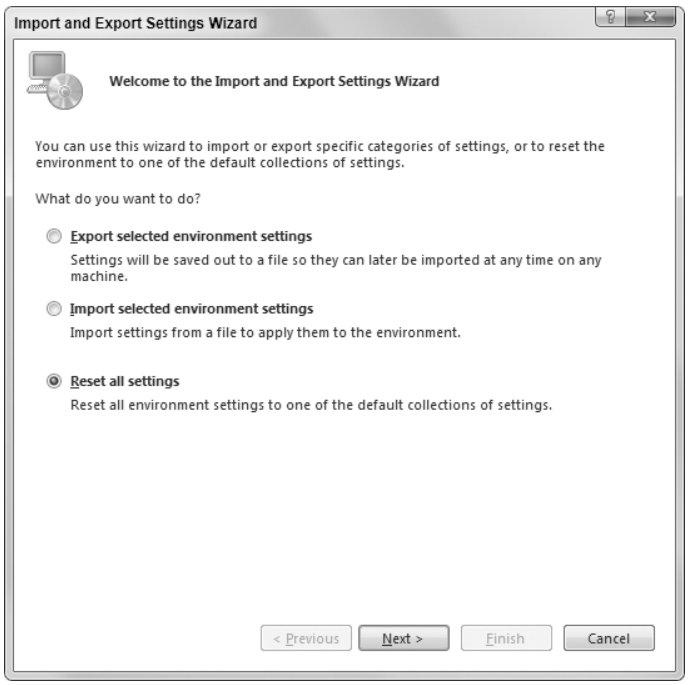


Рис. 2.1. *Сброс настроек VS*

После этого щелкните на кнопке Next (Далее) и укажите, хотите ли вы сохранить существующие параметры, прежде чем продолжить. Если вы уже выполняли какие-нибудь настройки, то, пожалуй, стоит сделать это; иначе щелкните на кнопке No (Нет), а затем снова на кнопке Next. В следующем диалоговом окне выберите вариант Visual C# Development Settings, как показано на рис. 2.2. Опять-таки, доступные пункты могут варьироваться.

После этого щелкните на кнопке Finish (Готово), чтобы изменения вступили в силу.

Схема расположения компонентов в среде VS является полностью настраиваемой, но вполне годится и предложенная по умолчанию. В случае выбора опции Visual C# Development Settings она будет выглядеть так, как показано на рис. 2.3.

Весь код отображается в главном окне, в котором по умолчанию при запуске VS содержится полезная страница Start Page (Начало работы). Это окно может содержать много документов на отдельных вкладках, и можно легко переключаться между различными файлами, щелкая на их именах. Кроме того, оно может отображать разрабатываемые графические интерфейсы, обычные текстовые файлы, HTML-код и различные встроенные в VS инструменты. Все это вы узнаете по мере прочтения книги.

Над главным окном находятся панели инструментов и меню VS. Здесь могут размещаться несколько панелей инструментов с возможностями от сохранения и загрузки файлов до сборки и запуска проектов, и даже отладки элементов управления. Они также будут описаны ниже.

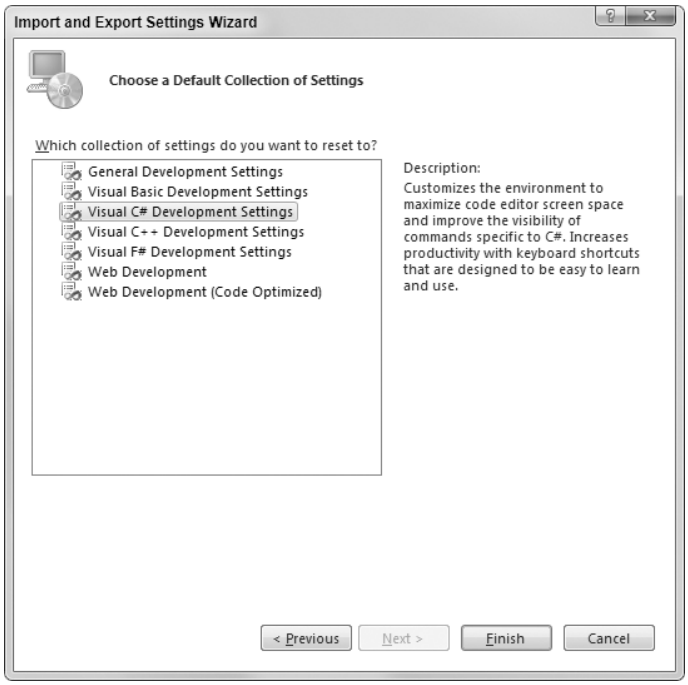


Рис. 2.2. *Выбор варианта* Visual C# Development Settings

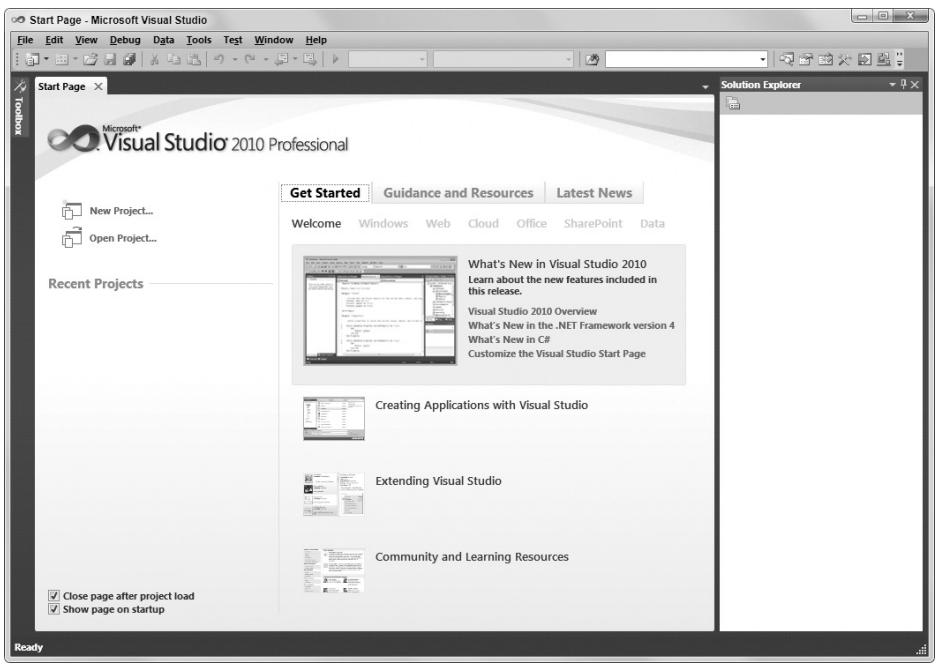


Рис. 2.3. *Стандартная компоновка VS*

Ниже даны краткие описания основных средств, которыми придется пользоваться чаще всего.

- Панель Toolbox (Панель инструментов). Появляется при наведении курсора на вкладку с соответствующим названием. Предоставляет доступ, в частности, к элементам пользовательского интерфейса для приложений Windows. Рядом с этой вкладкой может отображаться и вкладка Server Explorer (Проводник серверов), доступная через пункт меню View⇨Server Explorer (Вид⇨Проводник серверов) и включающая различные дополнительные возможности, вроде доступа к источникам данных, настройкам сервера, службам и т.д.
- Окно Solution Explorer (Проводник решений). В этом окне отображается информация о загруженном в данный момент *решении*. Как было сказано в предыдущей главе, в терминологии VS решением называется один или более проектов вместе с их конфигурациями. Окно Solution Explorer содержит различные представления входящих в состав решения проектов: какие файлы в них содержатся, и что содержится в этих файлах.
- Окно Properties (Свойства). Располагается непосредственно под окном Solution Explorer. На рис. 2.3 оно не показано потому, что появляется только во время работы над проектом (или после выбора пункта меню View⇨Properties Window (Вид⇨Окно свойств)). Это окно дает более подробное представление содержимого проекта и позволяет осуществлять дополнительную настройку его отдельных элементов. Например, оно позволяет изменить внешний вид кнопки на форме Windows.
- Окно Error List (Список ошибок). Это окно тоже не показано на снимке экрана, но является чрезвычайно важным. Открыть его можно выбором пункта меню View⇨Error List (Вид⇨Список ошибок). В нем отображаются ошибки, предупреждения и другая касающаяся проекта информация. Содержимое этого окна постоянно обновляется, хотя некоторая информация появляется в нем только при компиляции проекта.

Может показаться, что даже этих основных средств чересчур много, но скоро вы привыкнете. Уже при создании первого примера проекта вам придется воспользоваться многими из только что описанных элементов.

Среда VS может отображать и многие другие окна, как информационные, так и функциональные. Многие из них могут разделять экранное пространство с теми окнами, что были перечислены выше, а переключаться между ними можно с помощью вкладок. Некоторые из этих окон используются ниже в этой книге, а некоторые вы обнаружите самостоятельно при более детальном изучении среды VS.

Visual C# 2010 Express Edition

В случае VCE волноваться об изменении параметров не нужно. Очевидно, что данный продукт не будет применяться для программирования на Visual Basic, поэтому нет и соответствующих параметров. При первом запуске VCE появляется экран, очень похожий на то, что предлагается в VS (рис. 2.4).

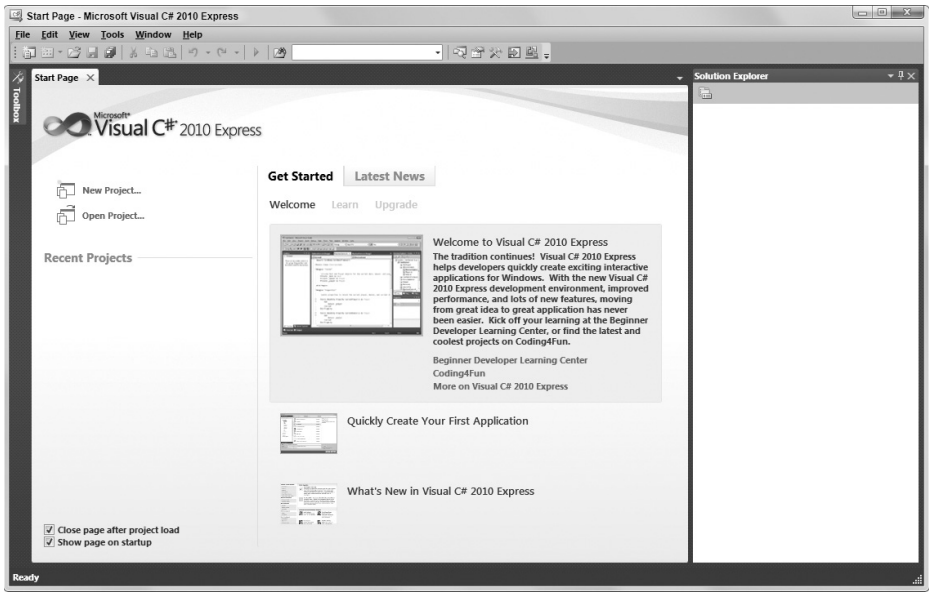


Рис. 2.4. *Экран, появляющийся при первом запуске VCE*

Консольные приложения

В настоящей книге мы будем регулярно пользоваться консольными приложениями, особенно поначалу, поэтому в следующем практическом занятии приводится пошаговое описание создания такого простого приложения — как в среде VS, так и в среде VCE.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание простого консольного приложения

1. Создайте новый проект консольного приложения, выбрав в меню File (Файл) пункт New⇨Project (Создать⇨Проект), если используется VS, или пункт New Project (Новый проект), если применяется VCE, как показано на рис. 2.5 и 2.6.

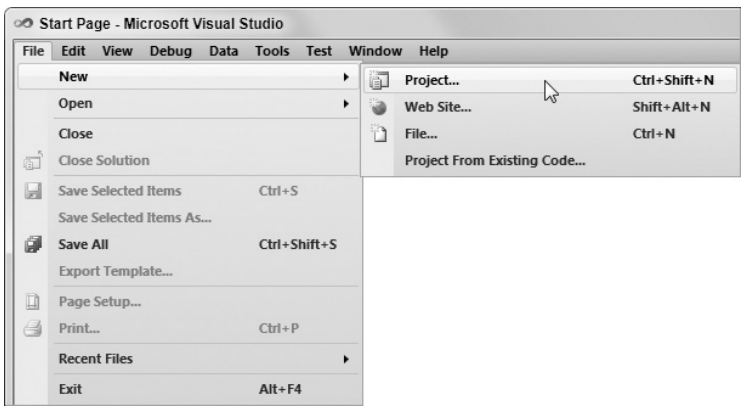


Рис. 2.5. *Создание нового проекта в VS*

2. В VS в появившейся панели Project Types (Типы проектов) выберите узел Visual C#, а в панели Templates (Шаблоны) — тип проекта Console Application (Консольное приложение), как показано на рис. 2.7. В VCE просто выберите в панели Templates (Шаблоны) тип Console Application (Консольное приложение), как показано на рис. 2.8. В VS измените путь в текстовом поле Location (Местоположение) на `C:\BegVCSharp\Chapter02` (если такой каталог не существует, он будет автоматически создан). Далее, как в VS, так и в VCE, оставьте без изменений предложенный по умолчанию текст в текстовом поле Name (Имя) (`ConsoleApplication1`), а также все остальные параметры (см. рис. 2.7 и 2.8)

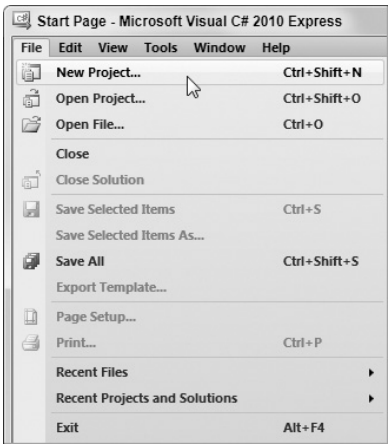


Рис. 2.6. *Создание нового проекта в VCE*

3. Щелкните на кнопке OK.
4. Если вы используете VCE, после инициализации проекта щелкните на кнопке Save All (Сохранить все) в панели инструментов или выберите пункт Save All (Сохранить все) в меню File (Файл), а потом укажите в поле Location (Местоположение) каталог `C:\BegVCSharp\Chapter02` и щелкните на кнопке OK.
5. После инициализации проекта добавьте в файл, появившийся в основном окне, следующие строки кода:

```
namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
     // Вывод текста на экран.
     Console.WriteLine("The first app in Beginning C# Programming!");
                       // Первое приложение на C#!
     Console.ReadKey();
    }
  }
}
```

Фрагмент кода ***ConsoleApplication1\Program.cs***

6. Выберите в меню Debug (Отладка) пункт Start Debugging (Начать отладку). После этого на экране должно появиться окно, показанное на рис. 2.9.
7. Нажмите любую клавишу, чтобы выйти из приложения (возможно, перед этим придется щелкнуть на окне консоли, чтобы оно получило фокус).

В VS такое окно появится только в том случае, если была выбрана опция Visual C# Developer Settings, как описано выше в данной главе. Если, к примеру, была выбрана не эта опция, а Visual Basic Developer Settings, тогда отобразится пустое окно консоли, а выходные данные приложения появятся в окне с заголовком Immediate (Непосредственный вывод). Кроме того, в этом случае не работает метод `Console.ReadKey()`, и возникнет ошибка. При наличии такой проблемы устраните ее на период работы с примерами, предлагаемыми в настоящей книге, применив опцию Visual C# Developer Settings — после этого полученные результаты должны совпадать с показанными здесь.

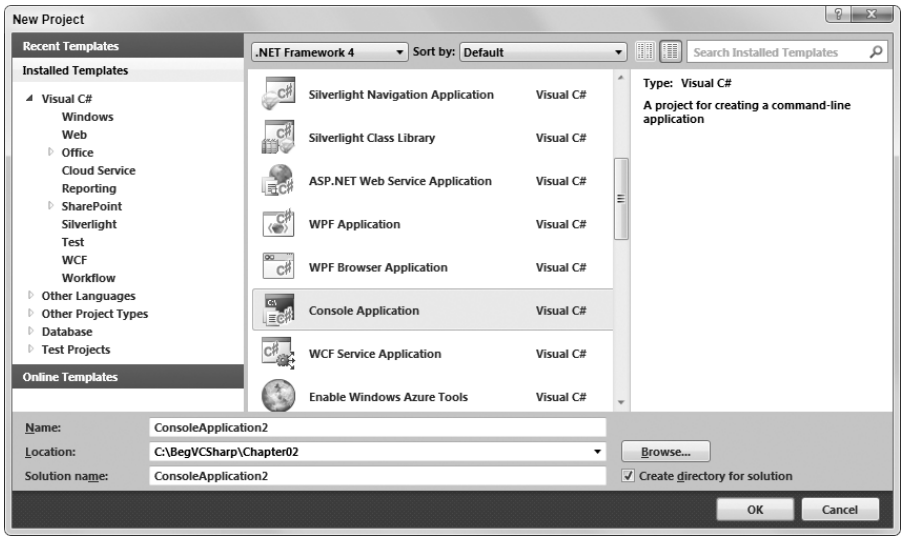


Рис. 2.7. *Создание консольного приложения в VS*

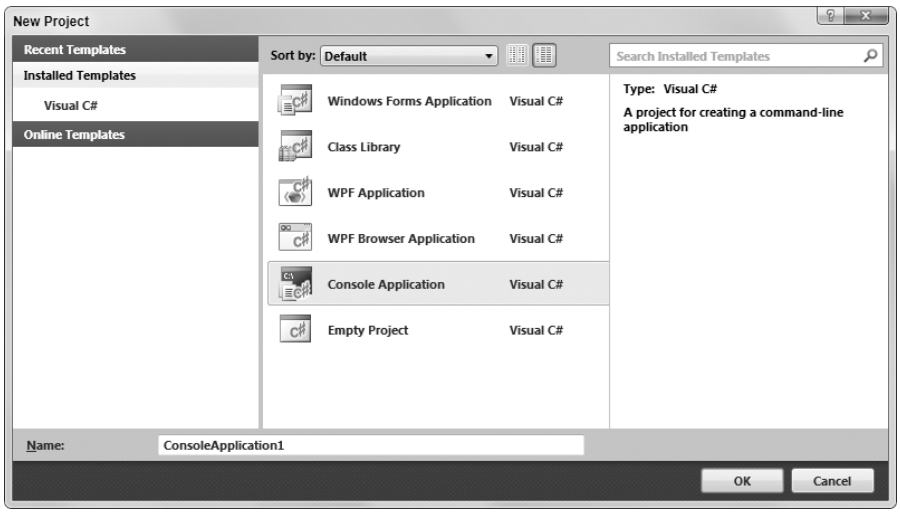


Рис. 2.8. *Создание консольного приложения в VCE*

Рис. 2.9. *Окно отладки приложения*

Если проблема не исчезнет, выберите пункт меню Tools⇨Options (Сервис⇨Параметры) и в открывшемся окне, в разделе Debugging (Отладка), снимите отметку с флажка Redirect all Output Window text to the Immediate Window (Перенаправлять весь текст из окна вывода в окно Immediate), как показано на рис. 2.10.

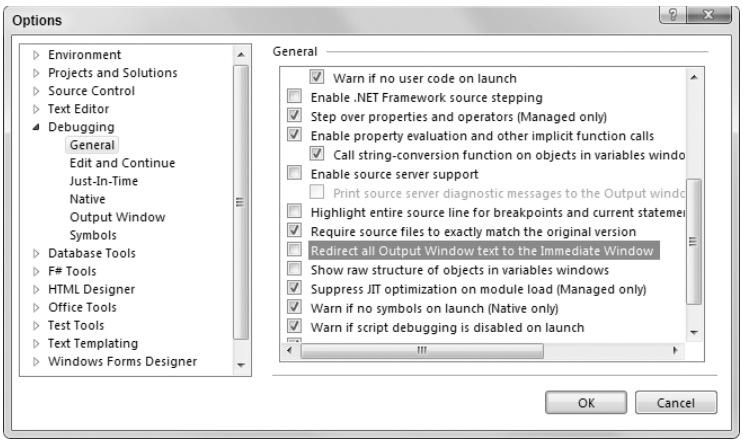


Рис. 2.10. *Флажок* Redirect all Output Window text to the Immediate Window

Описание работы

Пока мы не будем анализировать использованный код, поскольку сейчас важно просто освоить средства разработки для создания работоспособного кода. Очевидно, что как VS, так и VCE выполняют многое за разработчика, упрощая процесс компиляции и выполнения кода. Вообще-то есть несколько способов выполнения даже этих простейших шагов: например, новый проект можно создать и с помощью описанного пункта меню, и нажатием комбинации клавиш <Ctrl+Shift+N>, и щелчком на соответствующем значке в панели инструментов.

Компилировать и выполнять код тоже можно несколькими способами. Например, помимо уже описанного пункта меню Debug⇨Start Debugging, можно также использовать клавиатурную комбинацию (<F5>) или значок в панели инструментов. Код можно запустить на выполнение и без отладки, выбрав пункт меню Debug⇨Start Without Debugging (Отладка⇨Запустить без отладки) (или нажав <Ctrl+F5>). А можно просто откомпилировать проект, не запуская его (с отладкой или без), выбрав пункт меню Build⇨Build Solution (Сборка⇨Собрать решение) или нажав клавишу <F6>. Запустить проект на выполнение без отладки или выполнить сборку можно и с помощью значков в панели инструментов, хотя по умолчанию их там нет. После компиляции код можно выполнить, запустив созданный файл `.exe` из проводника Windows или командной строки. Для этого откройте окно командной строки, перейдите в каталог `C:\BegVCSharp\Chapter02\ConsoleApplication1\ConsoleApplication1\bin\Debug\`, введите **`ConsoleApplication1`** и нажмите клавишу <Enter>.

В последующих примерах, когда будут попадаться указания типа "создайте новый проект консольного приложения" или "выполните код", вы можете избрать для выполнения этих шагов любой способ. Весь код следует запускать с включенным режимом отладки, если не указано обратное. Кроме того, термины "выполнить" и "запустить" используются в этой книге как взаимозаменяемые, а в следующих за примерами объяснениях всегда предполагается, что вы вышли из созданного в примере приложения.

Работа консольных приложений завершается сразу же, как только заканчивается выполнение их кода, а это означает, что в случае запуска этих приложений непосредственно из IDE-среды вы можете и не увидеть результаты. Для устранения этой проблемы в предыдущем примере код должен был перед завершением ожидать нажатия клавиши с помощью следующей строки:

```
Console.ReadKey();
```

Этот прием будет еще не раз применяться в последующих примерах. Теперь, при наличии созданного проекта, можно более близко ознакомиться с некоторыми областями среды разработки.

Проводник решений

Окно Solution Explorer (Проводник решений) располагается в правом верхнем углу экрана. В VS и VCE оно выглядит одинаково (как и все остальные рассматриваемые в настоящей главе окна, если не сказано противоположное). По умолчанию это окно автоматически скрывается, но его можно пристыковать к краю экрана, щелкнув на значке с изображением канцелярской кнопки, когда оно видимо. Окно Solution Explorer располагается на экране вместе с еще одним полезным окном под названием Class View (Представление классов) — его можно увидеть, выбрав пункт меню View⇨Class View (Вид⇨Представление классов). На рис. 2.11 показаны оба этих окна со всеми развернутыми узлами (переключаться между пристыкованными окнами можно щелчками на вкладках с соответствующими названиями в нижней части окна).

В VCE окно Class View доступно только при включении параметра Expert Settings, который доступен с помощью пункта меню Tools⇨Settings⇨Expert Settings (Сервис⇨Настройки⇨Параметры для экспертов).

В данном случае в окне Solution Explorer отображаются файлы, из которых состоит проект `ConsoleApplication1`. Помимо файла `Program.cs`, в который вы добавляли код, показан еще один кодовый файл `AssemblyInfo.cs` и несколько ссылок.

Все файлы кода C# имеют расширение `.cs`.

Про файл `AssemblyInfo.cs` мы поговорим позже. В нем содержится дополнительная информация о проекте, которая пока нас не волнует.

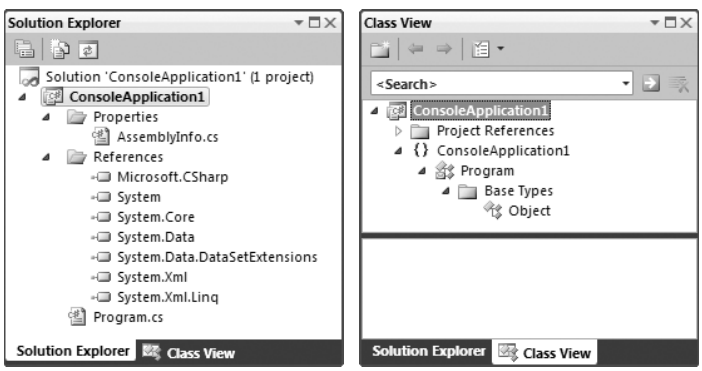


Рис. 2.11. *Окна Solution Explorer и Class View*

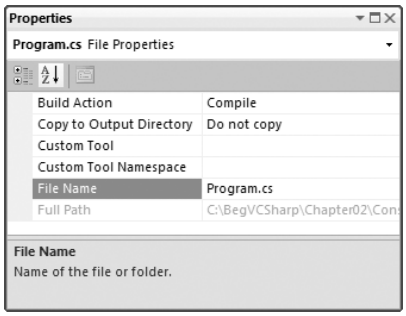


Рис. 2.12. *Внешний вид окна* Properties *после выбора файла* `Program.cs`

Это окно позволяет изменить код, который отображается в главном окне: можно дважды щелкнуть на файле с расширением `.cs`, либо щелкнуть на нем правой кнопкой мыши и выбрать в контекстном меню пункт View Code (Просмотреть код), либо выделить его и щелкнуть на соответствующей кнопке в панели инструментов, которая находится в верхней части окна. Здесь можно выполнять и другие операции с файлами, например, переименовывать их или удалять из проекта. В этом окне могут отображаться файлы других типов, вроде файлов ресурсов проекта (файлами ресурсов называются такие файлы, которые используются в проекте, но могут и не быть файлами C# — например, файлы растровых изображений или звуковые файлы). С ними также можно работать с помощью этого же самого интерфейса.

Узел References (Ссылки) содержит список используемых в проекте библиотек .NET. Об этом мы поговорим чуть позже; пока же вполне годятся стандартные ссылки. Второе окно — Class View — содержит другое отображение проекта, а именно структуру созданного кода. К нему мы тоже вернемся позже в этой книге; а пока годится то, что предлагается в окне Solution Explorer. Щелкая на файлах или других значках в этих окнах, можно заметить, как меняется содержимое окна Properties (рис. 2.12).

Окно Properties

В этом окне (если оно не открыто, выберите пункт меню View⇨Properties Window (Вид⇨Окно свойств)) отображается дополнительная информация о любом элементе, выбранном в окне над ним. К примеру, то, что показано на рис. 2.12, выводится в случае выбора в проекте файла `Program.cs`. В этом окне также отображается информация и о других выбираемых элементах, например, о компонентах пользовательского интерфейса (как будет показано в разделе “Приложения Windows Forms”).

Зачастую изменения, которые вносятся в окне Properties, непосредственно влияют на код проекта: добавляются новые строки или изменяются существующие. В некоторых проектах на настройку элементов в этом окне приходится тратить не меньше времени, чем на ручное изменение кода.

Окно Error List

Пока окно Error List (Список ошибок), доступное через пункт меню View⇨Error List (Вид⇨Список ошибок), не представляет интереса, поскольку в приложении `ConsoleApplication1` ошибок нет. Однако в действительности это очень полезное окно. Чтобы убедиться в этом, попробуйте удалить точку запятой из какой-нибудь строки, которая была добавлена в предыдущем разделе. Тут же в окне Error List появится примерно такое сообщение, как показано на рис. 2.13.

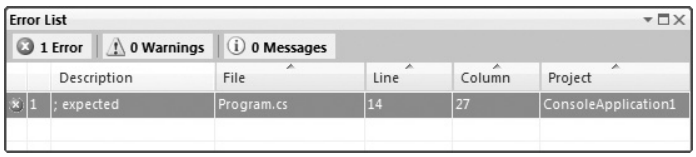


Рис. 2.13. *Сообщение об ошибке в окне* Error List

Кроме того, после этого скомпилировать проект уже не получится.

В главе 3, где будет рассматриваться синтаксис языка C#, вы узнаете, что символы точки с запятой встречаются в коде очень часто — в конце большинства строк.

Это окно помогает устранять ошибки в коде, поскольку следит за тем, что требуется для успешной компиляции проектов. Если дважды щелкнуть на сообщении об ошибке, курсор переместится в позицию этой ошибки в коде (если исходный файл, в котором содержится ошибка, не был открыт, он откроется автоматически) — это ускоряет поиск и исправление ошибок. Кроме того, ошибки в коде подчеркиваются волнистыми линиями красного цвета, благодаря которым легко видеть все проблемные места.

Местоположение ошибки указывается с помощью номера строки. По умолчанию в текстовом редакторе VS номера строк не отображаются, но это такая возможность, которую стоит включить. Для этого установите флажок Line Numbers (Номера строк) в окне Options (Параметры), доступном через пункт меню Tools⇨Options (Сервис⇨Параметры). Он находится в категории Text Editor⇨C#⇨General (Текстовый редактор⇨Язык C#⇨Общие), как показано на рис. 2.14.

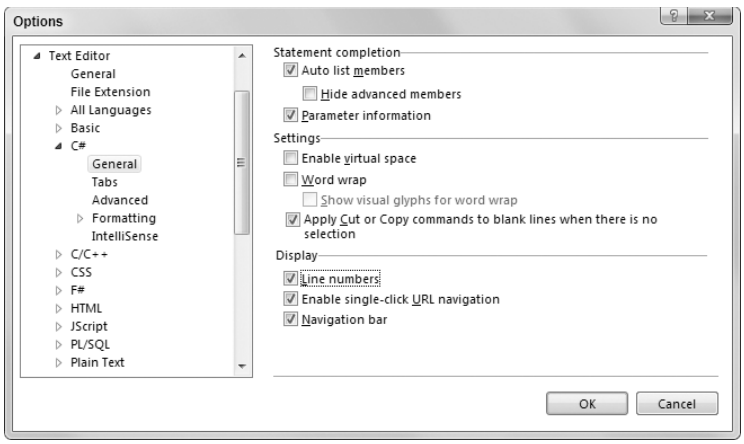


Рис. 2.14. *Флажок* Line Numbers

Чтобы этот параметр стал доступен в VCE, потребуется выбрать настройку Show All Settings *(Показывать все параметры); сам перечень параметров немного отличается от показанного на рис. 2.14.*

В этом диалоговом окне имеется много полезных параметров, и некоторые из них еще придется использовать ниже в этой книге.

Приложения Windows Forms

Часто легче продемонстрировать работу кода, если он является частью приложения Windows, а не через окно консоли или окно командной строки. Для создания такого приложения необходимо собрать пользовательский интерфейс, используя готовые блоки.

В следующем практическом занятии показаны лишь простейшие шаги, которые позволят создать и запустить приложение Windows, не вдаваясь в детали того, что оно делает на самом деле. Позже приложения Windows будут рассматриваться более подробно.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание простого приложения Windows Forms

1. Создайте (в VS или VCE) новый проект типа Windows Forms Application в том же каталоге, что и раньше (`C:\BegVCSharp\Chapter02`, и если вы используете VCE, сохраните в нем проект после создания) с именем по умолчанию `WindowsFormsApplication1`. Если вы используете VS и первый проект еще открыт, то для создания нового решения нужно выбрать вариант Create New Solution (Создать новое решение). Все описанные параметры показаны на рис. 2.15 и 2.16.

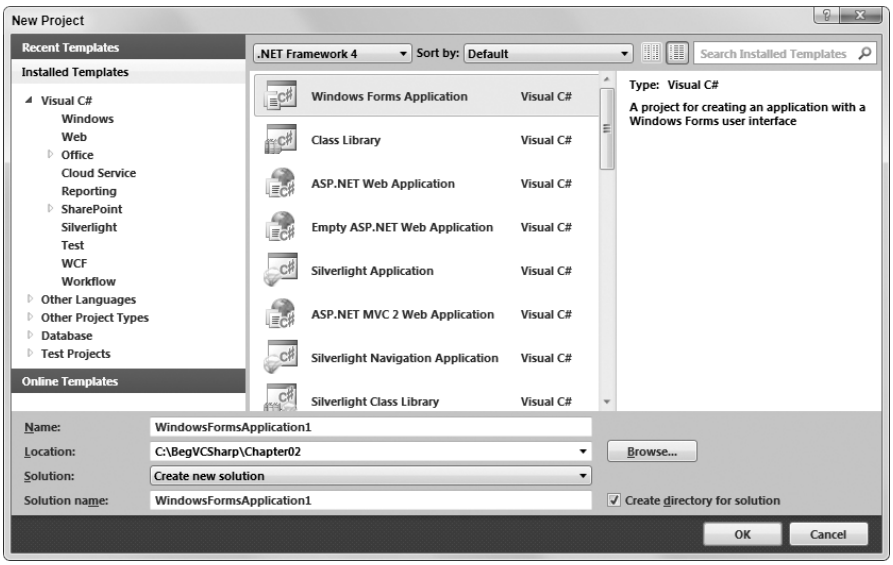


Рис. 2.15. *Создание приложения Windows Forms в VS*

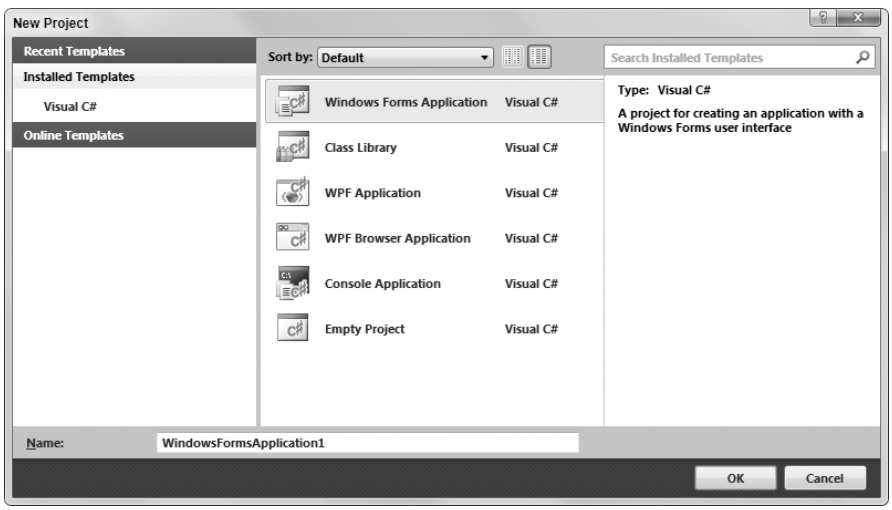


Рис. 2.16. *Создание приложения Windows Forms в VCE*

2. Щелкните на кнопке OK для создания проекта. После этого на экране должна появиться пустая форма Windows. Переместите курсор мыши на панель Toolbox в левой части экрана, наведите его на элемент Button (Кнопка) во вкладке All Windows Forms (Все формы Windows) и дважды щелкните на нем, чтобы добавить кнопку на главную форму приложения (`Form1`).
3. Дважды щелкните на только что добавленной кнопке.
4. После этого откроется файл `Form1.cs` с шаблонным кодом C#. Измените его, как показано ниже (для краткости здесь приведена лишь часть шаблонного кода):

Рис. 2.17. *Приложение `WindowsFormsApplication1` в действии*

```
private void button1_Click(object sender, EventArgs e)
{
  MessageBox.Show("The first Windows app in the book!");
                  // Первое приложение Windows в книге!
}
```

Фрагмент кода ***WindowsFormsApplication1\Form1.cs***

5. Запустите приложение.
6. Щелкните на (единственной) кнопке — появится диалоговое окно с сообщением, показанное на рис. 2.17.
7. Щелкните на кнопке OK, и затем выйдите из приложения, щелкнув на крестике в правом верхнем углу окна, как обычно в стандартных приложениях Windows.

Описание работы

И снова IDE-среда выполнила большую часть работы самостоятельно и позволила создать функциональное Windows-приложение с минимальными усилиями. Созданное приложение ведет себя точно так же, как и другие окна: его можно перемещать, изменять размеры, сворачивать и т.п. Писать специально код для этих операций не нужно — все и так работает. То же и с добавленной кнопкой. После двойного щелчка на ней IDE-среда решила, что вам понадобится написать код, который будет выполняться при щелчке пользователем на этой кнопке в работающем приложении. Теперь нужно лишь ввести этот код, получив все связанные с выполнением щелчка возможности задаром.

Конечно, приложения Windows не ограничиваются простыми формами с кнопками. Посмотрите на панель, с которой взят элемент `Button` — там имеется множество строительных блоков для пользовательского интерфейса, часть которых может быть вам знакома. Большинство из них будут использованы в данной книге, и вы увидите, что все они просты в применении и могут сэкономить разработчику массу времени и усилий.

Код этого приложения, содержащийся в файле `Form1.cs`, выглядит не намного сложнее кода из предыдущего раздела, как, впрочем, и код в остальных файлах в окне Solution Explorer. Значительная часть сгенерированного кода по умолчанию скрыта — ведь он связан с расположением элементов управления на форме, и поэтому его можно просматривать в основном окне в режиме визуального конструктора. Это просто визуальное представление кода компоновки. Кнопка является лишь примером элемента управления, который можно использовать подобно остальным блокам пользовательского интерфейса, доступным в разделе Windows Forms панели Toolbox.

Рассмотрим более подробно взятую в качестве примера элемента управления кнопку. Вернитесь в режим конструктора формы, воспользовавшись соответствующей вкладкой в главном окне, и щелкните один раз на кнопке, чтобы выделить ее. После этого в окне Properties (Свойства) в правом нижнем углу экрана появятся свойства этой кнопки (у элементов управления, как и у файлов, есть свои свойства). Проверьте, что приложение в данный момент находится в незапущенном состоянии, прокрутите окно свойств, пока не появится свойство `Text` со значением "`button1`", и измените его значение на "`Click Me`", как показано на рис. 2.18.

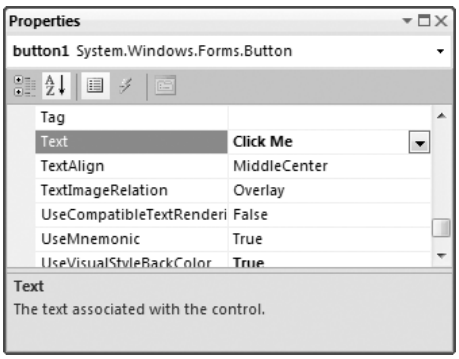


Рис. 2.18. *Изменение значения свойства* `Text`

Текст на кнопке, расположенной на форме `Form1` также должен измениться.

Для кнопок доступно множество свойств, от простых свойств для установки цвета и размера кнопки и до более сложных наподобие параметров привязки к данным. Как было упомянуто в предыдущем примере, изменение свойств часто приводит к непосредственным изменениям в коде, и данный случай тоже не является исключением. Однако если вернуться в режим просмотра кода `Form1.cs`, то никаких изменений в коде видно не будет.

Чтобы увидеть эти изменения, понадобится заглянуть в уже упоминавшийся скрытый код. Для просмотра файла, содержащего этот код, сначала раскройте в окне Solution Explorer узел `Form1.cs`, и станет виден узел `Form1.Designer.cs`. Чтобы просмотреть содержимое этого файла, дважды щелкните на нем.

На первый взгляд в этом коде ничего, что отражало бы изменение в свойстве кнопки. Объясняется это тем, что разделы кода на C#, которые отвечают за компоновку и форматирование элементов управления на форме, являются скрытыми (ведь если имеется графическое представление результатов, вряд ли потребуется просматривать код).

Для этого в VS и VCE используется система *организации кода*. То, как она работает, можно увидеть на рис. 2.19.

Слева от кода (рядом с номерами строк, если активизировано их отображение), расположены серые линии и квадратики с символами + и – внутри. Эти квадратики предназначены для разворачивания и сворачивания разделов кода. Ближе к концу файла имеется один такой квадратик со знаком + и прямоугольник напротив него внутри основного тела кода с надписью Windows Form Designer generated code (Код, сгенерированный конструктором форм Windows). Эта надпись, по сути, говорит о том, что имеется некий код, сгенерированный VS автоматически, о котором разработчику знать необязательно. Но при желании его все-таки можно просмотреть и увидеть результат изменения в свойстве `Text` кнопки. Для этого достаточно просто щелкнуть на квадратике с символом +. После этого весь скрытый код станет видимым, и где-то внутри него вы увидите следующую строку:

```
this.button1.Text = "Click Me";
```

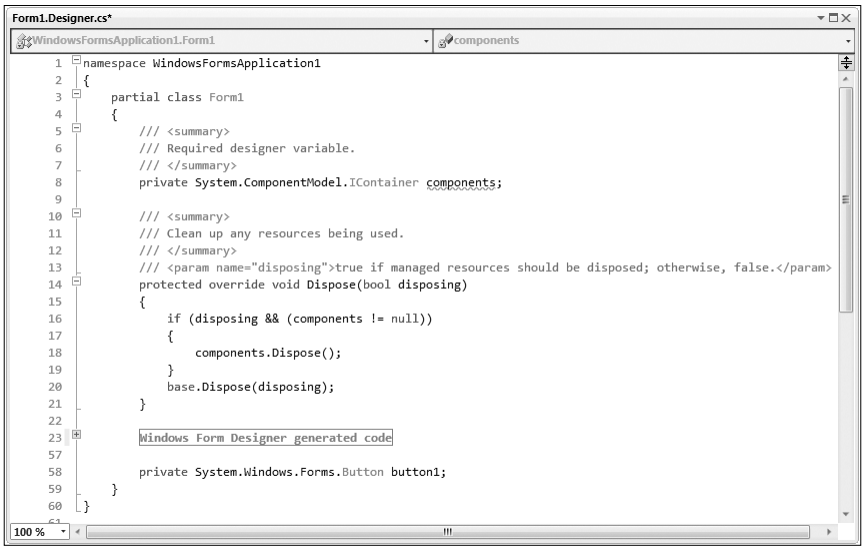


Рис. 2.19. *Организация кода*

На синтаксис, который здесь используется, пока не нужно обращать внимание. Главное, что здесь появился тот самый текст, который был введен в окне Properties.

Такой механизм организации кода очень удобен при написании кода, т.к. позволяет разворачивать и сворачивать и многие другие разделы, а не только скрытые по умолчанию. Точно так же, как просмотр содержания позволяет быстро получить общее представление о содержимом книги, просмотр ряда свернутых разделов кода может значительно упростить навигацию по объемному коду на C#.

Резюме

В настоящей главе были описаны средства, которыми неоднократно придется пользоваться в оставшейся части книги. Здесь был приведен краткий обзор сред разработки Visual Studio 2010 и Visual C# 2010 Express Edition и показано их применение для создания консольных приложений и приложений Windows. Консольные приложения более просты, но их достаточно для изучения основ программирования на C#. Приложения Windows сложнее, но зато более наглядны и понятны для тех, кто привык работать в среде Windows (а таких все же большинство).

Теперь, когда вы умеете создавать простые приложения, можно переходить непосредственно к изучению самого языка C#. В следующих главах речь пойдет сначала о базовом синтаксисе языка C# и структуре программ, затем о более сложных методах объектно-ориентированного программирования, а после этого — о том, как с помощью C# можно получить доступ ко всем возможностям, имеющимся в .NET Framework.

В последующих главах, если не указано обратное, все инструкции относятся к среде VCE, хотя, как было показано в настоящей главе, их несложно адаптировать под VS, поэтому вы можете применять любую IDE-среду, какая вам больше нравится или к которой имеете доступ.

Что вы узнали в этой главе

Тема	Основные концепции
Параметры Visual Studio 2010	Для проработки материала данной книги при первом запуске или сбросе параметров нужно установить режим разработки на C#.
Консольные приложения	Консольные приложения — это простые приложения для работы в окне командной строки, которые применяются в данной книге для иллюстрации описываемых приемов. Создавайте новые консольные приложения на основе шаблона Console Application, который доступен при создании нового проекта в VS или VCE. Для запуска проекта в отладочном режиме используйте пункт меню Debug⇨Start Debugging или нажмите клавишу <F5>.
Окна IDE-среды	Содержимое проекта отображается в окне Solution Explorer. Свойства выбранного элемента отображаются в окне Properties. Ошибки выводятся в окне Error List.
Приложения Windows Forms	Приложения Windows Forms — это приложения, имеющие внешний вид стандартного графического приложения со значками, которые позволяют развернуть, свернуть и закрыть приложение. Они создаются на основе шаблона Windows Forms в диалоговом окне New Project.

3

Переменные и выражения

В ЭТОЙ ГЛАВЕ...

- Базовый синтаксис C#
- Переменные и их применение
- Выражения и их применение

Для эффективного использования C# важно понимать, что на самом деле происходит при создании компьютерной программы. Пожалуй, наиболее простое определение компьютерной программы — это последовательность операций, обрабатывающих данные. Это действительно так даже в случае наиболее сложных примеров, вроде обширных многофункциональных приложений Windows (например, комплекта Microsoft Office). Обычно пользователям приложений этого совершенно не видно, но внутри компьютера все происходит именно так. Для более наглядной иллюстрации возьмем такую часть компьютера, как монитор. То, что мы видим на мониторе, зачастую кажется таким знакомым, и трудно представить, что это не "кинофильм", а нечто иное. На самом деле изображение на мониторе является лишь представлением данных, которые на физическом уровне являются просто набором запрятанных где-то в памяти нулей и единиц. Любое выполняемое на экране действие, вроде перемещения курсора мыши, щелчка на значке или ввода текста в окне текстового редактора, приводит к перемещению данных в памяти.

Можно взять и более простой пример — приложение-калькулятор. Оно получает данные в виде чисел и выполняет над этими числами различные математические операции примерно как в случае использования бумаги и карандаша, но только гораздо быстрее.

Поскольку действие компьютерных программ основывается на выполнении операций с данными, это означает, что нужен какой-то способ для хранения этих данных, и нужны методы для работы с ними. Эти две функции обеспечиваются с помощью, соответственно, *переменных* и *выражений*. О том, что это означает, и рассказывается в настоящей главе — как в общем, так и подробно.

Однако первым делом необходимо ознакомиться с базовым синтаксисом, который применяется в программировании на C#, поскольку без контекста изучать способы работы с переменными и выражениями в языке C# не получится.

Базовый синтаксис C#

По внешнему виду код C# похож на код C++ и Java. Поначалу этот синтаксис может показаться запутанным, ведь он мало похож на естественные языки. Однако после ознакомления с миром программирования на C# вы поймете, что применяемый стиль вполне разумен и позволяет писать понятный код без особых усилий.

В отличие от компиляторов некоторых других языков, компиляторы C# не обращают внимания на пустые места в коде — символы пустой строки, возврата каретки и табуляции (все вместе они называются *пробельными символами*). Это предоставляет разработчику огромную свободу в плане форматирования кода, хотя следование определенным правилам, конечно же, может сделать код более удобным для чтения.

Код на C# состоит из ряда *операторов*, каждый из которых оканчивается символом точки с запятой. Поскольку пробелы игнорируются, в одной строке могут размещаться сразу несколько операторов, но для удобочитаемости все-таки принято после символов точки с запятой добавлять возврат каретки, чтобы каждый оператор располагался в отдельной строке. Однако вполне допустимо (и довольно распространено) использовать операторы, занимающие несколько строк кода.

C# является *языком с блочной структурой*, т.е. все операторы относятся к какому-то *блоку* кода. В каждом блоке, который отделяется от остальной части кода с помощью фигурных скобок ({ и }), может содержаться любое количество операторов, в том числе и ни одного. Обратите внимание, что после фигурных скобок точка с запятой не требуется.

Например, простой блок кода на C# может иметь следующий вид:

```
{
  <строка_кода_1, оператор_1>;
  <строка_кода_2, оператор_2>
    <строка_кода_3, оператор_2>;
}
```

Здесь разделы <строка_кода_x, оператор_y> не являются фрагментами кода C#, а просто представляют собой метки-заполнители, указывающие, где должны находиться C#-операторы. В данном случае вторая и третья строки кода являются частью одного и того же оператора, поскольку в конце второй строки нет символа точки с запятой.

В следующем простом примере уже используются *отступы* для повышения наглядности самого кода C#. Это вполне стандартный прием, и по умолчанию среда VS выполняет его автоматически. Обычно для каждого блока кода используется свое количество отступов, т.е. своя величина смещения вправо. Блоки кода могут быть *вложенными* один в другой, т.е. одни блоки могут содержать другие блоки, и в таком случае *вложенные* блоки, соответственно, смещаются вправо на большее количество отступов:

```
{
  <строка_кода_1>;
  {
    <строка_кода_2>;
    <строка_кода_3>;
  }
  <строка_кода_4>;
}
```

Кроме того, строки, которые являются продолжением предыдущей строки, обычно также дополнительно сдвигаются вправо, как третья строка в первом примере.

Правила, которые VCE применяет для форматирования кода, можно увидеть в диалоговом окне Options *(Параметры), которое открывается выбором пункта меню* Tools⇨Options *(Сервис⇨Параметры). Этих правил очень много, и все они содержатся в различных подкатегориях узла* Text Editor⇨C#⇨Formatting *(Текстовый редактор⇨C#⇨Форматирование). Большинство из них касается тех частей C#, которые еще не рассматривались, но вы можете вернуться к ним позже и настроить в соответствии со своим стилем. На всякий случай сообщаем, что в настоящей книге все фрагменты кода приведены в том виде, который они имели бы при стандартных параметрах форматирования.*

Разумеется, такой стиль ни в коем случае не является обязательным. Однако если его не придерживаться, то уже очень скоро вы поймете, что иначе код может стать очень запутанным.

В коде на C# часто встречаются *комментарии*. Строго говоря, комментарии не являются кодом C#, но зато они замечательно с ним сосуществуют. Их смысл понятен без объяснений: они позволяют добавлять в код описательный текст на обычном языке, который компилятором игнорируется. При создании длинных разделов кода очень полезно оставлять в нем примечания о выполняемых там операциях, вроде “эта строка кода предлагает пользователю ввести число” или “этот раздел кода написал Вася”.

В C# комментарии можно вставлять двумя способами: либо с помощью специальных маркеров в начале и в конце комментария, либо с помощью одного (другого) маркера, который означает, что “остальная часть данной строки является комментарием”. Последний способ представляет собой исключение из упомянутого ранее правила об игнорировании компиляторами C# символов возврата каретки, но это особый случай.

Для обозначения комментариев первым способом в начале комментария ставятся символы /*, а в конце — символы */. Эти символы могут находиться как на одной и той же строке, так и на разных, и тогда все находящиеся между ними строки тоже считаются комментарием. Единственной комбинацией символов, которая невозможна в теле комментария, является */, потому что она воспринимается как маркер конца комментария. Вот примеры допустимых комментариев:

```
/* Это комментарий */
/* И это...
                ... тоже комментарий! */
```

А с таким комментарием возникнут проблемы:

```
/* Комментарии часто заканчиваются символами "*/"  */
```

Здесь конец комментария (символы, идущие после `"*/"`) будут восприниматься как код C#, из-за чего и будут возникать ошибки.

Второй вид комментариев начинается с комбинации символов `//`. После них можно вводить все, что угодно, главное — уместить все в одной строке. Ниже показан пример корректного комментария:

```
// Это другой вид комментария.
```

А вот следующий вариант неверен, поскольку вторая строка будет интерпретироваться не как комментарий, а как код C#:

```
// И это тоже,
   а вот этот фрагмент уже нет.
```

Такой способ комментирования удобно применять для пояснения операторов, поскольку он позволяет размещать операторы и комментарии в одной строке:

```
<оператор>; // Объяснение оператора
```

Ранее было сказано, что существует только два способа комментирования кода C#, но существует еще и третий способ, который, в принципе, является разновидностью синтаксиса `//`. Это однострочные комментарии, начинающиеся не с двух, а с трех символов слеша:

```
/// Специальный комментарий
```

Обычно такие комментарии игнорируются компилятором, подобно все остальным комментариям, но среду VS можно настроить так, чтобы при компиляции проекта она извлекала текст таких комментариев и создавала отформатированный специальным образом текстовый файл, который затем можно преобразовать в документацию. Для такого преобразования нужно, чтобы комментарии были составлены по правилам XML-документации — эта тема выходит за рамки настоящей книги, но при наличии времени с ней стоит ознакомиться.

Очень *важным* моментом, на который обязательно нужно обратить внимание в C#-коде, является его *чувствительность к регистру символов*. В отличие от некоторых других языков, в языке C# код обязательно нужно вводить в правильном регистре, поскольку из-за одной прописной буквы вместо строчной проект невозможно будет откомпилировать. Например, вот строка кода из главы 2:

```
Console.WriteLine("The first app in Beginning C# Programming!");
```

Она понятна C#-компилятору, потому что все буквы в команде `Console.WriteLine()` указаны в правильном регистре. Но ни одна из приведенных ниже строк работать не будет:

```
console.WriteLine("The first app in Beginning C# Programming!");
CONSOLE.WRITELINE("The first app in Beginning C# Programming!");
Console.Writeline("The first app in Beginning C# Programming!");
```

Во всех этих строках использован неправильный регистр, поэтому C#-компилятор не понимает, чего от него хотят. К счастью, как вы скоро узнаете, VCE очень помогает при вводе кода, и в большинстве случаев догадывается (насколько может догадываться компьютерная программа) о намерениях программиста. По мере ввода кода она предлагает команды, которые он может использовать, и старается исправлять ошибочный регистр символов.

Структура простого консольного приложения на C#

А теперь детальнее рассмотрим пример консольного приложения из главы 2 (ConsoleApplication1) и немного разберемся в его структуре. Ниже показан его код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      // Вывод текста на экран.
      Console.WriteLine("The first app in Beginning C# Programming!");
      Console.ReadKey();
    }
  }
}
```

Здесь присутствуют все рассмотренные в предыдущем разделе синтаксические элементы: и точки с запятой, и фигурные скобки, и комментарии, и соответствующие отступы.

Пока для нас наиболее важен следующий фрагмент:

```
static void Main(string[] args)
{
  // Вывод текста на экран.
  Console.WriteLine("The first app in Beginning C# Programming!");
  Console.ReadKey();
}
```

Именно этот код и выполняется при запуске приложения. Если говорить точнее, то при запуске выполняется блок кода, заключенный в фигурные скобки. Строка комментария, как и положено, не делает ничего; она просто поясняет код. Две остальные строки кода выводят некоторый текст в окно консоли и ожидают ответа; как именно они это делают, нас пока не касается.

Теперь посмотрим, как добиться организации кода, о которой рассказывалось в предыдущей главе. Обозначим с помощью ключевых слов #region и #endregion начало и конец раздела кода, который нужно сделать сворачиваемым. Например, код, сгенерированный для приложения ConsoleApplication1, можно изменить так:

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

#endregion
```

Это позволит сворачивать этот код в одну строку и снова разворачивать его позже при необходимости. Смысл операторов using и следующего сразу за ними оператора namespace будет объяснен в конце этой главы.

Любое ключевое слово, которое начинается с символа #, является директивой препроцессора, а не ключевым словом языка C#. Другие ключевые слова, кроме описанных здесь `#region` и `#endregion`, могут быть довольно сложными и имеют очень специфическое назначение. Эта одна из тем, изучением которых можно заняться самостоятельно после прочтения настоящей книги.

На остальной код в этом примере пока не нужно обращать внимания, потому что темой первых нескольких глав является базовый синтаксис C#, а точный способ, которым выполнение приложения доходит до вызова метода `Console.WriteLine()`, сейчас не важен. Позже смысл этого дополнительного кода станет понятен.

Переменные

Как уже было сказано, переменные предназначены для хранения данных. По сути, к переменным в памяти компьютера можно относиться как к коробкам на полке. В коробки можно класть какие-то вещи и затем снова доставать их оттуда, или просто заглядывать в них и проверять, есть ли там что-нибудь. То же самое и с переменными: в них можно помещать данные и затем при необходимости либо извлекать их оттуда, либо проверять их наличие.

Хотя все данные в компьютере, в сущности, имеют одинаковый вид (наборы нулей и единиц), переменные бывают разных видов, которые называются *типами*. Если вспомнить аналогию с коробками, то коробки могут быть разных форм и размеров, из-за чего некоторые вещи могут умещаться только в коробках определенного вида. Объясняется это тем, что для разных типов данных нужны разные методы их обработки, а ограничение переменных конкретными типами позволяет избегать их смешивания. Например, не стоит обрабатывать набор нулей и единиц, представляющий цифровое изображение, таким же образом, как и звуковой файл.

Чтобы использовать переменные, их нужно *объявлять*. Это означает, что им необходимо назначить *имя* и *тип*. После объявления их можно использовать в качестве единиц хранения для данных того типа, который задан при объявлении.

В синтаксисе C# для объявления переменной нужно просто указать ее тип и имя:

```
<тип> <имя>;
```

При попытке использовать переменную, которая не была объявлена, код не будет компилироваться, но в этом случае компилятор точно сообщит причину, и особых проблем не будет. При попытке использовать переменную без присваивания ей значения тоже возникнет ошибка, хотя компилятор распознает и ее.

Количество типов, которые можно использовать, практически бесконечно. Дело в том, что программист может определять собственные типы переменных для хранения каких угодно сложных данных. Однако существует ряд таких типов данных, которые используются очень часто, вроде переменной, позволяющей хранить числа. Поэтому о таких простых предопределенных типах необходимо знать каждому.

Простые типы

К простым типам относятся типы вроде числовых и булевских (`true` или `false`) значений, которые выступают в роли основных строительных блоков в любом приложении. В отличие от сложных типов, простые типы не могут иметь ни дочерних типов, ни атрибутов. Большинство из доступных простых типов являются числовыми, что на первый взгляд может показаться странным — ведь для хранения чисел вроде необходим один тип?

Причина такого изобилия числовых типов связана с механизмом хранения чисел в памяти компьютера в виде последовательностей нулей и единиц. В случае целочисленных

значений просто берется ряд *битов* (отдельных разрядов, которыми могут быть 0 или 1), и число представляется в двоичном (бинарном) формате. Переменная, хранящая *N* битов, позволяет представлять любое число в диапазоне от 0 до $2^N - 1$. Любые числа, большие этого значения, не умещаются в такой переменной.

Пусть, например, имеется переменная, способная хранить 2 бита. Тогда соответствие между целыми числами и представляющими их битами будет таким:

```
0 = 00
1 = 01
2 = 10
3 = 11
```

Если нужен больший диапазон чисел, потребуется больше битов (например, 3 бита позволяют хранить числа от 0 до 7).

Неизбежным результатом такой системы является то, что для хранения любого вообразимого числа понадобится бесконечное количество битов, которое никак не сможет уместиться в памяти обычного ПК. А если взять какое-то большое количество битов для хранения почти любого числа, то использовать все эти биты для переменной, в которой требуется хранить, к примеру, только числа от 0 до 10, будет неэффективно (потому что память будет тратиться зря).

Вместо этого лучше применять различные целочисленные типы, способные хранить разные диапазоны чисел и занимающие разные объемы памяти (вплоть до 64 битов). Все эти типы перечислены в табл. 3.1.

Каждый из этих типов соответствует одному из стандартных типов, определенных в .NET Framework. Как уже рассказывалось в главе 1, именно такое применение стандартных типов обеспечивает функциональную совместимость языков. Имена, используемые для типов в C#, являются псевдонимами для типов, которые определены в .NET Framework. В табл. 3.1 имена типов представлены в виде, который они имеют в библиотеке .NET Framework.

Таблица 3.1. Основные целочисленные типы

Тип	Псевдоним для	Допустимые значения
`sbyte`	`System.SByte`	Целые числа от –128 до 127
`byte`	`System.Byte`	Целые числа от 0 до 255
`short`	`System.Int16`	Целые числа от –32768 до 32767
`ushort`	`System.UInt16`	Целые числа от 0 до 65535
`int`	`System.Int32`	Целые числа от –2147483648 до 2147483647
`uint`	`System.UInt32`	Целые числа от 0 до 4294967295
`long`	`System.Int64`	Целые числа от –9223372036854775808 до 9223372036854775807
`ulong`	`System.UInt64`	Целые числа от 0 до 18446744073709551615

Символы `u` в начале имен некоторых типов являются сокращением от слова *unsigned* (без знака), т.е. хранить отрицательные числа в переменных этих типов нельзя, что и указано в столбце “Допустимые значения” таблицы.

Разумеется, также необходимо хранить значения *с плавающей точкой*, т.е. значения, которые не являются целыми числами. Для них имеются переменные трех следующих типов: `float`, `double` и `decimal`. Первые два позволяют хранить значения с плавающей точкой в виде $\pm m \times 2^e$, причем допустимые значения для `m` и `e` у них отличаются. А тип `decimal` позволяет хранить значения в виде $\pm m \times 10^e$. Все эти три типа переменных показаны в табл. 3.2 вместе с допустимыми значениями для `m` и `e` и предельными значениями в формате с плавающей точкой.

Таблица 3.2. Типы переменных для хранения чисел с плавающей точкой

Тип	Псевдоним для	Мин. m	Макс. m	Мин. e	Макс. e	Приблиз. мин. значение	Приблиз. макс. значение
`float`	`System.Single`	0	2^{24}	–149	104	1.5×10^{-45}	3.4×10^{38}
`double`	`System.Double`	0	2^{53}	–1075	970	5.0×10^{-324}	1.7×10^{308}
`decimal`	`System.Decimal`	0	2^{96}	–26	0	1.0×10^{-28}	7.9×10^{28}

Помимо числовых типов, доступны и другие простые типы, которые перечислены в табл. 3.3.

Таблица 3.3. Нечисловые простые типы

Тип	Псевдоним для	Допустимые значения
`char`	`System.Char`	Одиночный символ Unicode, хранимый в виде целого числа от 0 до 65535
`bool`	`System.Boolean`	Булево значение — `true` либо `false`
`string`	`System.String`	Последовательность символов

Обратите внимание на отсутствие ограничения для количества символов, которые могут храниться в `string`, поскольку этот тип может использовать различные объемы памяти.

Булевский тип `bool` является одним из наиболее часто используемых типов переменных в C#. Вообще-то аналогичные типы часто применяются и в других языках. Наличие переменной, которая может содержать либо значение `true`, либо значение `false`, играет важную роль при выполнении логики приложения. В качестве простого примера представьте количество вопросов, на которые можно ответить с помощью "да" или "нет" (т.е. посредством `true` или `false`). Выполнение сравнений между значениями переменных и проверка правильности входных данных — лишь два примера программного применения булевских переменных, которые очень скоро будут рассмотрены более подробно.

Теперь, после ознакомления со всеми простыми типами, рассмотрим один небольшой пример объявления и использования переменных. В следующем практическом занятии приводится простой код, который объявляет две переменные, присваивает им значения и затем выводит эти значения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование переменных простых типов

1. Создайте новое консольное приложение с именем `Ch03Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter03`.
2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
   int myInteger;
   string myString;
   myInteger = 17;
   myString = "\"myInteger\" is";   // myInteger равно
   Console.WriteLine("{0} {1}.", myString, myInteger);
   Console.ReadKey();
}
```

Фрагмент кода **Ch03Ex01\Program.cs**

3. Выполните этот код. На рис. 3.1 показан результат, который должен получиться.

Рис. 3.1. *Результат выполнения приложения* `Ch03Ex01`

Описание работы

Добавленный код выполняет три действия:

- объявляет две переменные;
- присваивает значения этим двум переменным;
- выводит значения этих двух переменных на консоль.

Объявление переменных происходит в следующей части кода:

```
int myInteger;
string myString;
```

В первой строке объявляется переменная типа `int` с именем `myInteger`, а во второй — переменная типа `string` с именем `myString`.

Именование переменных подчиняется определенным правилам; указывать просто любую последовательность символов нельзя. Более подробно об этом будет рассказано ниже в этой главе, в разделе "Именование переменных".

В следующих двух строках кода выполняется присваивание значений:

```
myInteger = 17;
myString = "\"myInteger\" is";
```

Здесь переменным присваиваются два фиксированных значения (называемые *литеральными* значениями) с помощью *операции присваивания* = (об операциях см. ниже, в разделе "Выражения"). Переменной `myInteger` присваивается целочисленное значение `17`, а переменной `myString` — строка `"myInteger" is` (вместе с кавычками). При таком присваивании строковых литеральных значений строки обязательно должны заключаться в двойные кавычки. Из-за этого включение в саму строку некоторых символов — например, символов двойных кавычек — может привести к появлению проблем. Поэтому такие символы следует *литерализовать*: вместо необходимого символа или символов подставляется специальная последовательность символов, называемая *управляющей последовательностью*. В данном примере для литерализации символа двойной кавычки применяется последовательность `\"`:

```
myString = "\"myInteger\" is";
```

Без нее строка кода выглядела бы следующим образом:

```
myString = ""myInteger" is";
```

и компилятор сообщил бы об ошибке.

Еще одна ситуация, когда следует соблюдать осторожность при использовании разрывов строк — присваивание строковых литералов, т.к. компилятор C# отбрасывает все строковые литералы, занимающие более одной строки. Если нужно вставить в литерал символ конца строки, лучше использовать управляющую последовательность для символа возврата каретки `\n`. Например, после присваивания

```
myString = "Эта строка содержит\nсимвол разрыва строки.";
```

такая строка будет выводиться в окне консоли в виде двух отдельных строк:

```
Эта строка содержит
символ разрыва строки.
```

Все управляющие последовательности состоят из символа обратной косой черты (слеша), за которым следует один символ из небольшого специального набора (этот набор будет представлен ниже). Из-за того, что символ слеша применяется для этой цели, для него самого тоже существует управляющая последовательность — два следующих друг за другом слеша (`\\`).

Вернемся к рассматриваемому коду. В нем есть еще одна новая строка:

```
Console.WriteLine("{0} {1}.", myString, myInteger);
```

Она похожа на простой метод вывода текста на консоль, который применялся в первом примере, но только теперь здесь указаны переменные. Чтобы не забегать вперед, пока опустим рассмотрение всяких деталей и лишь скажем, что для вывода текста на консоль в первой части настоящей книги будет применяться именно такой прием. Внутри скобок содержатся:

- строка, подлежащая выводу;
- список разделенных запятыми переменных, значения которых должны вставляться в выводную строку.

На первый взгляд в выводимой строке — `"{0} {1}."` — нет никакого полезного текста. Однако на экране при выполнении кода отображается совсем другое. Дело в том, что данная строка на самом деле представляет собой шаблон, в который вставляется содержимое переменных. Каждая пара фигурных скобок в строке является заполнителем, вместо которого во время выполнения будет вставлено содержимое одной из указанных в списке переменных.

Каждый заполнитель (или строка форматирования) представляется в виде заключенного в фигурные скобки целого числа. Отсчет этих целых начинается с нуля с шагом 1. Общее количество заполнителей должно соответствовать количеству переменных, указанных в разделенном запятыми списке следом за строкой форматирования. При выводе текста на консоль каждый заполнитель заменяется значением соответствующей переменной. В рассматриваемом примере это означает, что `{0}` заменяется фактическим значением первой переменной, т.е. `myString`, а `{1}` — содержимым переменной `myInteger`.

Такой метод вывода текста в окно консоли будет применяться для вывода выходных данных из кода в последующих примерах. И, наконец, в рассматриваемом коде имеется уже знакомая вам строка для ожидания нажатия клавиши пользователем перед завершением работы:

```
Console.ReadKey();
```

Эта строка кода тоже пока подробно рассматриваться не будет, но она будет часто встречаться в последующих примерах. Пока достаточно запомнить, что она приостанавливает выполнение кода до нажатия любой клавиши.

Именование переменных

Как уже было сказано в предыдущем разделе, выбрать любую последовательность символов в качестве имени переменной нельзя. Однако все не так плохо, потому что система именования все равно является очень гибкой.

Основные правила по созданию имен переменных:

- Первым символом в имени переменной должна быть либо буква, либо символ подчеркивания (_), либо символ @.
- Последующими символами в имени переменной могут быть буквы, символы подчеркивания или цифры.

Кроме этого, существуют ключевые слова, которые имеют для компилятора C# особое значение, вроде уже знакомых вам ключевых слов `using` и `namespace`. В случае использования одного из таких слов компилятор выдаст сообщение, поэтому сильно беспокоиться об этом не стоит. Ниже приведены приемлемые имена для переменных:

```
myBigVar
VAR1
_test
```

А такие имена являются недопустимыми:

```
99BottlesOfBeer
namespace
It's-All-Over
```

Не забывайте, что язык C# чувствителен к регистру символов, так что при объявлении переменных запоминайте их точное написание. Их упоминание в программе с неверным регистром даже одной буквы приведет к невозможности компиляции. Еще одно следствие — можно иметь несколько переменных, имена которых отличаются только регистром символов. Например, все эти имена переменных различаются в C#:

```
myVariable
MyVariable
MYVARIABLE
```

Соглашения об именовании

Имена переменных используются *постоянно*, поэтому стоит посвятить немного времени, чтобы разобраться, какие имена следует применять. Только учтите, что это спорный вопрос. Постоянно появляются и исчезают разные системы, и многие разработчики горячо отстаивают преимущества своей личной системы.

До недавнего времени наиболее популярной считалась *венгерская нотация* (Hungarian notation). В ней в начале имен всех переменных добавляются строчные префиксы, означающие тип этих переменных. Например, если переменная имеет тип `int`, то в начале ее имени должен быть префикс `i` (или `n`), т.е. оно может выглядеть наподобие `iAge`. Применение этой системы позволяет с первого взгляда определять тип переменной.

Однако в современных языках вроде C# эта система сталкивается с трудностями. Для типов, которые пока были рассмотрены, пожалуй, еще можно подобрать одно- или двухбуквенные префиксы. Но из-за возможности создавать собственные типы и наличия в базовой библиотеке .NET Framework многих сотен таких более сложных типов этот подход становится непрактичным. Если над проектом работает нескольких людей, легко может случиться так, что каждый из них придумает свои разные запутанные префиксы, что может привести к катастрофическим последствиям.

Разработчики поняли, что гораздо лучше именовать переменные в соответствии с их назначением. При возникновении каких-либо сомнений тип переменной можно всегда легко выяснить. В VS и VCE для этого достаточно просто навести на имя переменной курсор мыши, и вскоре появится всплывающее окно с информацией о ее типе.

В настоящее время в пространствах имен .NET Framework используются две системы именования: *PascalCase* (стиль языка Pascal) и *camelCase* ("верблюжий" стиль). Обе они применяются к именам, которые состоят из нескольких записанных слитно слов, и обе требуют, чтобы все буквы в каждом слове имени были строчными, кроме первой, которая должна быть заглавной. В системе camelCase есть еще одно дополнительное правило: первое слово начинается со строчной буквы.

Таким образом, имена переменных, созданные по системе camelCase, могут выглядеть, к примеру, так:

```
age
firstName
timeOfDeath
```

А имена переменных, созданные по системе PascalCase, соответственно, так:

```
Age
LastName
WinterOfDiscontent
```

Для именования простых переменных пользуйтесь системой camelCase, а в случае более сложных переменных в Microsoft рекомендуют применять PascalCase. Раньше во многих прежних системах именования часто применялся символ подчеркивания, обычно для разделения слов в именах переменных, например: `yet_another_variable`. Сейчас это встречается нечасто (компилятору все равно, но для людей выглядит неуклюже).

Литеральные значения

В предыдущем практическом занятии были показаны два примера литеральных значений — `integer` и `string`. У других типов тоже могут быть соответствующие литеральные значения (см. табл. 3.4). Во многих из них используются *суффиксы*, т.е. добавление в конце литерального значения последовательности символов, которая указывает нужный тип. Некоторые литералы имеют по несколько типов, которые определяются во время компиляции самим компилятором на основании контекста (что тоже можно увидеть в табл. 3.4).

Таблица 3.4. Литералы

Тип (типы)	Категория	Суффикс	Пример / допустимые значения
`bool`	Булевский	Нет	`true` или `false`
`int`, `uint`, `long`, `ulong`	Целочисленный	Нет	`100`
`uint`, `ulong`	Целочисленный	`u` или `U`	`100U`
`long`, `ulong`	Целочисленный	`l` или `L`	`100L`
`ulong`	Целочисленный	`ul`, `uL`, `Ul`, `UL`, `lu`, `lU`, `Lu` или `LU`	`100UL`
`float`	Вещественный	`f` или `F`	`1.5F`
`double`	Вещественный	Нет, `d` или `D`	`1.5`
`decimal`	Вещественный	`m` или `M`	`1.5M`
`char`	Символьный	Нет	`'a'` или управляющая последовательность
`string`	Строковый	Нет	`"a...a"`, может включать управляющие последовательности

Строковые литералы

Ранее в этой главе уже было показано несколько управляющих последовательностей, которые могут использоваться в литералах `string`. В табл. 3.5 для справки перечислены все такие управляющие последовательности.

Таблица 3.5. Управляющие последовательности, используемые в строковых литералах

Управляющая последовательность	Означает	Unicode-значение символа
`\'`	Символ одиночной кавычки	`0x0027`
`\"`	Символ двойной кавычки	`0x0022`
`\\`	Символ обратной косой черты	`0x005C`
`\0`	Нулевой символ	`0x0000`
`\a`	Символ предупреждения (приводит к выдаче звукового сигнала)	`0x0007`
`\b`	Символ забоя	`0x0008`
`\f`	Переход на новую страницу	`0x000C`
`\n`	Переход на новую строку	`0x000A`
`\r`	Возврат каретки	`0x000D`
`\t`	Горизонтальная табуляция	`0x0009`
`\v`	Вертикальная табуляция	`0x000B`

В столбце "Unicode-значение символа" приведены шестнадцатеричные значения символов в наборе символов Unicode. Вообще-то любой символ Unicode можно указывать с помощью управляющей последовательности Unicode. Такие управляющие последовательности состоят из стандартного символа `\`, за которым идет символ `u` и четырехзначное шестнадцатеричное значение (как, например, четыре цифры после `x` в табл. 3.5).

Это означает, что следующие строки эквивалентны:

```
"Karli\'s string."
"Karli\u0027s string."
```

Очевидно, что управляющие последовательности Unicode дают гораздо большие возможности.

Строки можно также задавать *буквально* или *дословно* (verbatim) — в этом случае в строку включаются все содержащиеся между двумя двойными кавычками символы, вместе с символами конца строки и другими символами, которые иначе пришлось бы литерализовать. Единственным исключением является управляющая последовательность для символа двойных кавычек, которая должна обязательно указываться во избежание окончания строки. Для этого достаточно просто добавить перед строкой символ `@`:

```
@"Verbatim string literal."
```

Эту строку точно так же легко можно задать и без символа `@`, но в следующем случае без этого метода не обойтись:

```
@"A short list:
item 1
item 2"
```

Буквальные строки особенно удобны для имен файлов, т.к. они обычно содержат много символов обратной косой черты. В обычных строках придется постоянно добавлять по два символа обратной косой черты:

```
"C:\\Temp\\MyDir\\MyFile.doc"
```

Буквальные строки позволяют получать более читабельный результат. Например, предыдущая строка будет выглядеть так:

```
@"C:\Temp\MyDir\MyFile.doc"
```

Как будет показано позже в этой книге, в отличие от других знакомых вам типов, которые являются типами значения, строки относятся к ссылочным типам. Из этого, в частности, следует, что строкам можно присвоить также значение `null`*, означающее, что данная строковая переменная не указывает на строку (точнее, ни на что вообще).*

Объявление переменных и присваивание им значений

Вы уже знаете, что переменные объявляются просто путем указания их типа и имени:

```
int age;
```

После этого им с помощью операции = можно присваивать значения:

```
age = 25;
```

Помните: перед использованием переменные должны быть инициализированы. В качестве операции инициализации может применяться аналогичная операция присваивания.

В коде C# часто встречается и пара других приемов. Во-первых, одновременно можно объявить сразу нескольких переменных одного и того же типа, разделяя их имена запятыми:

```
int xSize, ySize;
```

Здесь `xSize` и `ySize` одновременно объявлены как переменные типа `int`.

Во-вторых, зачастую в коде C# при объявлении переменным сразу же присваиваются значения, что фактически означает объединение двух строк кода:

```
int age = 25;
```

Оба этих приема могут применяться и вместе:

```
int xSize = 4, ySize = 5;
```

Здесь объявлены две переменные — `xSize` и `ySize`, которым сразу же присваиваются разные значения. Учтите, что в строке

```
int xSize, ySize = 5;
```

выполняется инициализация только переменной `ySize`, а переменная `xSize` лишь объявляется, и перед использованием ее еще надо инициализировать.

Выражения

Теперь, когда вы умеете объявлять и инициализировать переменные, пора научиться работать с ними. В C# для этого предусмотрен ряд *операций*. Комбинируя эти операции с переменными и литеральными значениями (при использовании вместе с операциями все они называются *операндами*), можно создавать *выражения*, которые являются главными элементами вычислений.

Доступные операции бывают как простые, так и очень сложные, а часть из них вряд ли встречается за пределами математических приложений. К простым относятся все основные математические операции, вроде +, позволяющей складывать два операнда вместе; а к сложным — операции над содержимым переменных в двоичном формате. Еще имеются логические операции, предназначенные специально для работы с булевскими значениями, и операции присваивания наподобие =.

В настоящей главе основное внимание уделяется математическим операциям и операциям присваивания. Логические операции будут более подробно рассматриваться в следующей главе, которая посвящена изучению булевской логики в контексте управления исполнением программы.

Операции делятся на три категории:

- **унарные**, выполняемые над одним операндом;
- **бинарные**, выполняемые над двумя операндами;
- **тернарные**, выполняемые над тремя операндами.

Большинство операций относится к бинарным, несколько к унарным, и лишь одна, называемая условной операцией (conditional operator) — к категории тернарных (эта условная операция является логической, и потому будет рассматриваться в главе 4). Начнем с изучения математических операций, которые бывают как унарными, так и бинарными.

Математические операции

Всего существует пять простых математических операций, две из которых (+ и –) имеют как бинарную, так и унарную версию. Все они приведены в табл. 3.6 вместе с краткими примерами их использования и результатами, которые получаются в случае их применения к простым числовым типам (целые числа и числа с плавающей точкой).

Таблица 3.6. Простые математические операции

Операция	Категория	Пример	Результат
+	Бинарная	`var1 = var2 + var3;`	`var1` присваивается значение — результат сложения значений `var2` и `var3`
–	Бинарная	`var1 = var2 - var3;`	`var1` присваивается значение — результат вычитания значения `var3` из значения `var2`
*	Бинарная	`var1 = var2 * var3;`	`var1` присваивается значение — результат умножения значений `var2` и `var3`
/	Бинарная	`var1 = var2 / var3;`	`var1` присваивается значение — результат деления значения `var2` на значение `var3`
%	Бинарная	`var1 = var2 % var3;`	`var1` присваивается значение — остаток от деления значения `var2` на значение `var3`
+	Унарная	`var1 = +var2;`	`var1` присваивается значение `var2`
–	Унарная	`var1 = -var2;`	`var1` присваивается значение `var2`, умноженное на `-1`

Унарная операция + является немного странной, потому что никак не влияет на результат. Она не делает значения положительными, как можно было бы предположить: если `var` содержит значение `-1`, то и `+var` тоже равно `-1`. Однако она является общеизвестной операцией, и потому включена в C#. О наиболее полезной особенности этой операции будет рассказано позже, при изучении перегрузки операций.

В примерах применяются простые числовые типы, потому что в случае использования других простых типов результат может быть не очевиден. Например, какой результат следует ожидать при сложении двух булевских значений? В данном случае никакой, поскольку при попытке использовать операцию + (или любую другую математическую операцию) с переменными типа `bool` компилятор выдаст ошибку. Со сложением переменных `char` дело тоже обстоит не совсем просто. На самом деле они хранятся в виде чисел, поэтому сложение двух переменных даст в результате также число (а именно, типа `int`). Это пример *неявного преобразования*, о котором (как и о *явном преобразовании*) будет рассказано чуть позже в этой главе, потому что оно выполняется и в тех случаях, когда `var1`, `var2` и `var3` имеют разные типы.

А вот бинарная операция + для строковых переменных имеет смысл, как показано в табл. 3.7.

Таблица 3.7. Строковая операция сложения

Операция	Категория	Пример	Результат
+	Бинарная	`var1 = var2 + var3;`	`var1` присваивается значение — результат сцепления хранящихся в `var2` и `var3` строк

Никакая другая математическая операция со строками не работает.

Осталось ознакомиться еще с двумя операциями — унарными операциями инкремента и декремента, которые могут записываться либо непосредственно перед операндом, либо сразу после него. Результаты, получаемые в случае применения этих операций в простых выражениях, показаны в табл. 3.8.

Таблица 3.8. Операции инкремента и декремента

Операция	Категория	Пример	Результат
++	Унарная	`var1 = ++var2;`	`var1` присваивается значение `var2 + 1`. `var2` увеличивается на 1
--	Унарная	`var1 = --var2;`	`var1` присваивается значение `var2 - 1`. `var2` уменьшается на 1
++	Унарная	`var1 = var2++;`	`var1` присваивается значение `var2`. `var2` увеличивается на 1
--	Унарная	`var1 = var2--;`	`var1` присваивается значение `var2`. `var2` уменьшается на 1

Эти операции всегда приводят к изменению значения, хранящегося в их операнде:

- операция ++ всегда увеличивает значение операнда на единицу;
- операция -- всегда уменьшает значение операнда на единицу.

Отличия между результатами, сохраняемыми в `var1`, вытекают из того, что местоположение операции определяет момент ее действия. Запись одной из этих операций перед

операндом означает выполнение этой операции перед выполнением любых других вычислений, а после операнда — означает выполнение операции после всех остальных вычислений в выражении. Это заслуживает еще одного примера. Рассмотрим следующий код:

```
int var1, var2 = 5, var3 = 6;
var1 = var2++ * --var3;
```

Какое значение будет присвоено переменной `var1`? Перед вычислением выражения будет выполнена записанная перед переменной `var3` операция `--`, которая изменит ее значение с `6` на `5`. На операцию, записанную после переменной `var2`, можно не обращать внимания, поскольку она не будет выполняться, пока не завершится вычисление выражения, а это значит, что переменной `var1` будет присвоен результат умножения `5` на `5`, т.е. `25`.

Эти простые унарные операции оказываются весьма кстати в удивительно большом количестве ситуаций. Они, по сути, представляют собой сокращение выражений вроде

```
var1 = var1 + 1;
```

Подобные выражения применяются очень часто, особенно в *циклах*, и об этом более подробно речь пойдет в главе 4. В следующем практическом занятии демонстрируется использование математических операций, а также вводятся несколько других полезных концепций. Код предлагает ввести строку и два числа и затем выводит результаты выполнения над введенными числами ряда вычислений.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ Действия с переменными с помощью математических операций

1. Создайте новое консольное приложение по имени `Ch03Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter03`.

2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
  double firstNumber, secondNumber;
  string userName;
  Console.WriteLine("Enter your name:");         // Введите ваше имя
  userName = Console.ReadLine();
  Console.WriteLine("Welcome {0}!", userName);   // Добро пожаловать
  Console.WriteLine("Now give me a number:");    // Теперь введите число
  firstNumber = Convert.ToDouble(Console.ReadLine());
  Console.WriteLine("Now give me another number:");
                  // Теперь введите другое число
  secondNumber = Convert.ToDouble(Console.ReadLine());
  Console.WriteLine("The sum of {0} and {1} is {2}.", // Результат сложения
    firstNumber, secondNumber, firstNumber + secondNumber);
  Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
                  // Результат вычитания
    secondNumber, firstNumber, firstNumber - secondNumber);
  Console.WriteLine("The product of {0} and {1} is {2}.",
                  // Результат умножения
    firstNumber, secondNumber, firstNumber * secondNumber);
  Console.WriteLine("The result of dividing {0} by {1} is {2}.",
                  // Результат деления
    firstNumber, secondNumber, firstNumber / secondNumber);
  Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
                  // Остаток от деления
    firstNumber, secondNumber, firstNumber % secondNumber);
  Console.ReadKey();
}
```

Фрагмент кода ***Ch03Ex02\Program.cs***

3. Запустите полученный код. На экране должно появиться то, что показано на рис. 3.2.

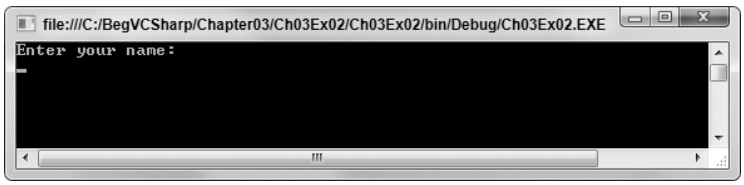


Рис. 3.2. *Выполнение приложения* `Ch03Ex02`

4. Введите свое имя и нажмите клавишу <Enter>, как показано на рис. 3.3.

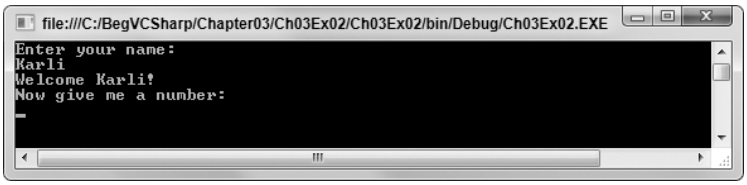


Рис. 3.3. *Ввод имени*

5. Введите число и нажмите клавишу <Enter>, затем введите еще одно число и снова нажмите <Enter>, как показано на рис. 3.4.

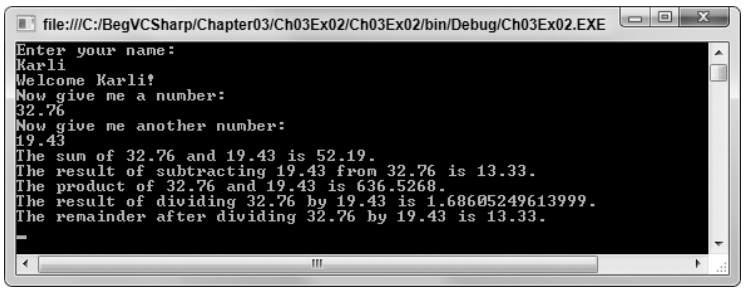


Рис. 3.4. *Ввод чисел*

Описание работы

Помимо математических операций, в этом коде также впервые представляются два важных понятия, с которыми вы будете постоянно встречаться:

- пользовательский ввод;
- преобразование типов.

Для пользовательского ввода применяется синтаксис, похожий на уже известную команду `Console.WriteLine()`, а именно, `Console.ReadLine()`. Эта команда предлагает пользователю ввести данные, которые затем сохраняются в переменной типа `string`:

```
string userName;
Console.WriteLine("Enter your name:");         // Введите ваше имя
userName = Console.ReadLine();
Console.WriteLine("Welcome {0}!", userName);   // Добро пожаловать
```

Данный код выводит содержимое присвоенной переменной `userName` прямо на экран. Кроме того, в этом примере выполняется считывание двух чисел. Данный процесс является немного более сложным, поскольку команда `Console.ReadLine()` генерирует строку, а требуется число. Здесь вступает в действие механизм *преобразования типов*. Более подробно о нем будет рассказываться в главе 5, а пока просто проанализируем код примера.

Сначала объявляются переменные, в которых должны сохраняться вводимые числа:

```
double firstNumber, secondNumber;
```

Далее выводится приглашение, и введенная строка `Console.ReadLine()` преобразуется в число `double` с помощью команды `Convert.ToDouble()`. Это число присваивается объявленной ранее переменной `firstNumber`:

```
Console.WriteLine("Now give me a number:");        // Теперь введите число
firstNumber = Convert.ToDouble(Console.ReadLine());
```

Синтаксис вполне прост и позволяет аналогично выполнять много других преобразований.

Потом точно так же вводится второе число:

```
Console.WriteLine("Now give me another number:");  // Теперь введите другое число
secondNumber = Convert.ToDouble(Console.ReadLine());
```

После этого производится вывод на экран результатов сложения, вычитания, умножения и деления двух полученных чисел, а также отображение остатка от деления, получаемого с помощью операции `%`:

```
Console.WriteLine("The sum of {0} and {1} is {2}.",     // Результат сложения
  firstNumber, secondNumber, firstNumber + secondNumber);

Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
                  // Результат вычитания
  secondNumber, firstNumber, firstNumber - secondNumber);

Console.WriteLine("The product of {0} and {1} is {2}.", // Результат умножения
  firstNumber, secondNumber, firstNumber * secondNumber);

Console.WriteLine("The result of dividing {0} by {1} is {2}.",
                  // Результат деления
  firstNumber, secondNumber, firstNumber / secondNumber);

Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
                  // Остаток от деления
  firstNumber, secondNumber, firstNumber % secondNumber);
```

Обратите внимание, что выражения, вроде `firstNumber + secondNumber` и т.д., передаются оператору `Console.WriteLine()` напрямую в виде параметра, без использования промежуточной переменной:

```
Console.WriteLine("The sum of {0} and {1} is {2}.",
  firstNumber, secondNumber, firstNumber + secondNumber);
```

Подобный синтаксис может существенно повысить наглядность кода и сократить его объем.

Операции присваивания

Пока что в этой главе встречалась только простая операция присваивания (=), и может показаться странным, что существуют и другие операции присваивания. Однако они действительно существуют, и они весьма удобны! Все операции присваивания, отличные от =, работают подобным образом. Как и операция =, все они выполняют присваивание значения переменной, которая находится в левой части, на основании операндов и операций, находящихся в их правой части. Все операции присваивания перечислены в табл. 3.9.

Таблица 3.9. Операции присваивания

Операция	Категория	Пример	Результат
=	Бинарная	`var1 = var2;`	`var1` присваивается значение `var2`
+=	Бинарная	`var1 += var2;`	`var1` присваивается значение — результат сложения `var1` и `var2`
-=	Бинарная	`var1 -= var2;`	`var1` присваивается значение — результат вычитания `var2` из `var1`
*=	Бинарная	`var1 *= var2;`	`var1` присваивается значение — результат умножения `var1` на `var2`
/=	Бинарная	`var1 /= var2;`	`var1` присваивается значение — результат деления `var1` на `var2`
%=	Бинарная	`var1 %= var2;`	`var1` присваивается значение — остаток от деления `var1` на `var2`

Все дополнительные операции приводят к изменению значения `var1`, т.е. строка кода вроде

```
var1 += var2;
```

выполняется точно так же, как и строка

```
var1 = var1 + var2;
```

Операция += может использоваться и со строками, точно так же, как операция +.

Применение этих операций, особенно в случае длинных имен переменных, может сделать код гораздо более понятным.

Порядок выполнения операций

При вычислении выражения каждая операция выполняется по очереди, но совсем не обязательно слева направо. В качестве простейшего примера рассмотрим следующее выражение:

```
var1 = var2 + var3;
```

Здесь операция + выполняется раньше, чем =. Однако бывают ситуации, когда приоритеты выполнения операций не столь очевидны, как, например, в следующем выражении:

```
var1 = var2 + var3 * var4;
```

В этом выражении первой будет выполняться операция *, за ней операция +, и только потом операция =. Такой порядок является общепринятым в математике и дает такой же результат, как и ручной арифметический подсчет.

Использование круглых скобок, как и в математике, позволяет управлять порядком выполнения операций; например, рассмотрим следующее выражение:

```
var1 = (var2 + var3) * var4;
```

Здесь первым будет вычисляться содержимое круглых скобок, т.е. операция + будет выполнена перед операцией *.

В табл. 3.10 приведены приоритеты уже знакомых нам операций, причем если приоритеты совпадают (как, например, у операций * и /), то операции выполняются слева направо.

Таблица 3.10. Приоритеты операций

Приоритет	Операции
Высший	++, -- (префиксные); +, - (унарные)
	*, /, %
	+ , -
	=, *=, /=, %=, +=, -=
Низший	++, -- (постфиксные)

Как уже было сказано, для переопределения порядка выполнения операций можно пользоваться круглыми скобками. Кроме того, учтите, что операции ++ и -- в постфиксной форме имеют, как видно из таблицы, низшие приоритеты только с концептуальной точки зрения. Они не влияют на результат, скажем, выражения присваивания, поэтому можно считать, что они обладают более высоким приоритетом, чем все остальные операции. Но поскольку они меняют значение своего операнда после вычисления выражения, проще считать их приоритеты такими, как указано в табл. 3.10.

Пространства имен

Сейчас удобно рассмотреть еще одну важную тему, а именно — *пространства имен* (namespace). Они являются в .NET своего рода способом предоставления контейнеров для кода приложения, позволяющих однозначно идентифицировать этот код и его содержимое. Кроме того, они применяются и для категоризации элементов в .NET Framework. Большую часть этих элементов составляют определения типов, наподобие определений тех простых типов, с которыми мы ознакомились в настоящей главе (`System.Int32` и т.д.).

По умолчанию C#-код содержится в *глобальном пространстве имен*. Это означает, что к содержащимся в этом коде элементам возможен доступ из любого другого кода в глобальном пространстве имен просто по их имени. Но можно воспользоваться ключевым словом `namespace` и явным образом определить пространство имен для блока кода, заключенного в фигурные скобки. В случае применения имен в коде за пределами такого пространства имен они должны обязательно *квалифицироваться*.

Квалифицированное имя — это имя, в котором содержится вся иерархическая информация. По сути, это значит, что при необходимости использовать в коде в одном пространстве имен имя, определенное в другом пространстве имен, нужно обязательно включать вместе с именем и ссылку на другое пространство имен. Для разделения уровней пространства имен используется символ точки (`.`):

```
namespace LevelOne
{
   // Код в пространстве имен LevelOne
   // Определение имени NameOne
}
// Код в глобальном пространстве имен
```

В этом коде определяется одно пространство имен — `LevelOne` — и одно имя в этом пространстве имен — `NameOne` (для общности описания сам код не приводится, а на месте определения находится комментарий). В коде внутри пространства имен `LevelOne` к этому имени можно обращаться просто как `NameOne`, т.е. квалифицировать его не обязательно. Однако в коде внутри глобального пространства имен потребуется квалифицированное имя `LevelOne.NameOne`.

По соглашению названия пространств имен обычно записываются в стиле языка Pascal.

Внутри любого пространства имен с помощью ключевого слова `namespace` можно определять вложенные пространства имен. Обращаться к ним нужно через их иерархию, а уровни иерархии разделяются точками. Лучше всего показать это на примере. Рассмотрим следующий код:

```
namespace LevelOne
{
   // Код в пространстве имен LevelOne

   namespace LevelTwo
   {
      // Код в пространстве имен LevelOne.LevelTwo
      // Определение имени NameTwo
   }
}
// Код в глобальном пространстве имен
```

К имени `NameTwo` в глобальном пространстве имен нужно обращаться как `LevelOne.LevelTwo.NameTwo`, в пространстве имен `LevelOne` — как `LevelTwo.NameTwo`, а в пространстве имен `LevelOne.LevelTwo` — как `NameTwo`.

Главное здесь то, что пространства имен однозначно идентифицируют определенные в них имена. Например, имя `NameThree` можно определить и в пространстве имен `LevelOne`, и в пространстве имен `LevelTwo`:

```
namespace LevelOne
{
   // Определение имени NameThree

   namespace LevelTwo
   {
      // Определение имени NameThree
   }
}
```

Здесь определены два отдельных имени `LevelOne.NameThree` и `LevelOne.LevelTwo.NameThree`, которые можно использовать независимо друг от друга.

После создания пространств имен для упрощения доступа к содержащимся в них именам можно применять оператор `using`. По сути, этот оператор означает: "Далее будут использоваться имена из этого пространства имен, поэтому не будем постоянно уточнять их". Например, следующий код указывает, что код в пространстве имен `LevelOne` должен иметь доступ к именам из пространства имен `LevelOne.LevelTwo` без уточнения:

```
namespace LevelOne
{
   using LevelTwo;

   namespace LevelTwo
   {
      // Определение имени NameTwo
   }
}
```

Теперь в пространстве имен `LevelOne` можно обращаться к переменной `LevelTwo.NameTwo` просто как `NameTwo`.

Однако в некоторых случаях, как в приведенном выше примере с `NameThree`, такой подход может привести к проблемам из-за конфликтов между идентичными именами в разных пространствах имен (т.е. при использовании такого имени код не будет компилиро-

ваться, а компилятор будет выдавать сообщение о неоднозначности). В подобных случаях в операторе `using` можно указывать *псевдоним* (alias) пространства имен:

```
namespace LevelOne
{
   using LT = LevelTwo;

   // Определение имени NameThree

   namespace LevelTwo
   {
      // Определение имени NameThree
   }
}
```

Благодаря этому код в пространстве имен `LevelOne` может обращаться к `LevelOne.NameThree` как `NameThree`, а к `LevelOne.LevelTwo.NameThree` как `LT.NameThree`.

Действие операторов `using` распространяется как на те пространства имен, в которых они содержатся, так и на любые вложенные пространства имен, которые могут в них содержаться. Приведенный выше код не позволяет применять в глобальном пространстве имен имя `LT.NameThree`. Однако если определить псевдоним следующим образом:

```
using LT = LevelOne.LevelTwo;

namespace LevelOne
{
   // Определение имени NameThree

   namespace LevelTwo
   {
      // Определение имени NameThree
   }
}
```

то имя `LT.NameThree` можно использовать в коде и глобального пространства имен, и пространства имен `LevelOne`.

Обратите внимание на еще один важный момент: сам по себе оператор `using` не предоставляет доступ к именам из другого пространства имен. Если код в пространстве имен не связан каким-то образом с проектом — например, с помощью определения в файле исходного кода проекта или в другом связанном с проектом коде — доступ к содержащимся в нем именам невозможен. Кроме того, в случае связывания содержащего пространство имен кода с проектом доступ к его именам будет возможен вне зависимости от того, используется оператор `using` или нет. Оператор `using` просто упрощает доступ к этим именам и может сократить объем кода, делая его более наглядным.

Теперь вернемся к приведенному в начале этой главы коду приложения `ConsoleApplication1` со следующими касающимися пространств имен строками:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   ...
}
```

Здесь четыре строки, начинающиеся с ключевого слова `using`, объявляют, что в последующем C#-коде будут использоваться пространства имен `System`, `System.Collections.Generic`, `System.Linq` и `System.Text` и что к ним возможен доступ без квалификации из всех пространств имен в данном файле. Пространство имен `System` является корневым для приложе-

ний .NET Framework и содержит все базовые функциональные возможности, необходимые для создания консольных приложений. Еще три пространства имен очень часто используются в консольных приложениях и потому указаны здесь просто так, на всякий случай.

После этого объявлено пространство имен `ConsoleApplication1`, предназначенное для кода самого приложения.

Резюме

В этой главе был предоставлен большой объем основополагающего материала, необходимого для создания практичных (хотя и простых) приложений на C#. Вы познакомились с основами синтаксиса C# и проанализировали код простого консольного приложения, который VS и VCE генерируют автоматически при создании проекта консольного приложения.

Большая часть этой главы была посвящена использованию переменных. Вы узнали, что такое переменные, как их можно создавать, как присваивать им значения, а также как работать с ними и значениями, которые они содержат. Попутно были продемонстрированы некоторые основные приемы взаимодействия с пользователем: вывод текста в консольном приложении и считывание вводимых пользователем данных. Для этого понадобились элементарные преобразования типов, но эта сложная тема будет более подробно рассмотрена в главе 5.

Кроме того, в этой главе было показано, как собирать операции и операнды в выражения, а также каким образом и в каком порядке они выполняются.

И, наконец, вы узнали о пространствах имен, которые очень важны для изучения дальнейшего материала в настоящей книге. Данная тема была пока освещена лишь в общих чертах, однако это дает основу для последующих обсуждений.

Пока все наше программирование сводилось к выполнению кода строка за строкой. В следующей главе вы узнаете, как повысить эффективность кода, управляя потоком выполнения с помощью циклов и условных ветвлений.

Упражнения

1. Как в следующем коде обратиться к имени `great` из кода в пространстве имен `fabulous`?

```
namespace fabulous
{
   // код в пространстве имен fabulous
}
namespace super
{
   namespace smashing
   {
      // определение имени great
   }
}
```

2. Какая из следующих последовательностей символов не является допустимым именем для переменной?

   а) `myVariableIsGood`

   б) `99Flake`

   в) `_floor`

   г) `time2GetJiggyWidIt`

   д) `wrox.com`

3. Является ли строка "supercalifragilisticexpialidocious" слишком большой для того, чтобы уместиться в переменной string? Если да, то почему?
4. Учитывая приоритеты операций, приведите шаги вычисления следующего выражения:

```
resultVar += var1 * var2 + var3 % var4 / var5;
```

5. Напишите консольное приложение, которое получает от пользователя четыре значения типа int и выводит их произведение. Подсказка: вспомните, что для преобразования введенного текста в тип double применялась команда Convert.ToDouble(); эквивалентная команда для преобразования данных из string в int выглядит как Convert.ToInt32().

Решения упражнений приведены в приложении A.

Что вы узнали в этой главе

Тема	Основные концепции
Базовый синтаксис языка C#	Язык C# чувствителен к регистру символов, и (почти) каждая строка кода оканчивается точкой с запятой. Для облегчения чтения строки продолжения операторов или вложенные блоки можно записывать с отступом. В код можно добавлять некомпилируемые комментарии с помощью синтаксиса // или /* ... */. Кроме того, для повышения наглядности кода отдельные блоки кода можно оформить в виде сворачиваемых областей.
Переменные	Переменные — это фрагменты данных, у которых есть имя и тип. .NET Framework определяет для обработки данных множество простых типов наподобие числовых и строковых (текстовых) типов. Перед использованием переменные необходимо объявить и инициализировать, причем это можно сделать вместе.
Выражения	Выражения составляются из операций и операндов, и операции выполняют действия над операндами. Имеются три вида операций — унарные, бинарные и тернарные — которые обрабатывают один, два и три операнда соответственно. Математические операции выполняют действия с числовыми значениями, а операции присваивания помещают результат выражения в переменную. У операций имеются фиксированные приоритеты, которые определяют порядок их выполнения в выражении.
Пространства имен	Все имена, определяемые в приложении .NET, в том числе и имена переменных, содержатся в некотором пространстве имен. Пространства имен образуют иерархии, и для доступа к ним часто необходимо уточнять имена в соответствии с содержащим их пространством имен.

4

Управление потоком выполнения

В ЭТОЙ ГЛАВЕ...

- Булевская логика и ее применение
- Способы реализации ветвлений
- Способы выполнения циклов

У всех приведенных до сих пор примеров кода C# имелась одна общая черта. В каждом случае выполнение программы переходило от одной строки к следующей сверху вниз, без всяких пропусков. Если бы все приложения работали подобным образом, то разработчики были бы очень ограничены в своих возможностях. В этой главе описываются два способа управления потоком выполнения программы (т.е. порядком выполнения строк C#-кода): *ветвление* (branching) и *циклы* (looping). Ветвление позволяет выполнять код в зависимости от результата вычисления — к примеру, только в том случае, если значение переменной `myVal` меньше 10. Циклы обеспечивают многократное выполнение одних и тех же операторов — либо определенное количество раз, либо до достижения некоторого условия.

Оба этих способа основаны на использовании *булевской логики*. В предыдущей главе тип `bool` упоминался, но никак не использовался. В этой главе он будет использоваться постоянно, поэтому глава начинается с описания того, что представляет собой булевская логика, чтобы в последующих примерах ее применение не вызывало никаких вопросов.

Булевская логика

Тип `bool`, с которым мы познакомились в предыдущей главе, может содержать только одно из двух значений — `true` или `false`. Этот тип часто применяется для сохранения результата какой-нибудь операции, чтобы в дальнейшем действовать в зависимости от этого результата. В частности, тип `bool` используется для хранения результата *сравнений*.

Историческая справка: основой булевской логики являются труды английского математика Джорджа Буля (George Boole), жившего в середине девятнадцатого столетия.

Например, предположим (как уже предлагалось во вводной части данной главы), что нужно, чтобы код выполнялся, только если значение переменной `myVal` меньше 10. Для этого нужно как-то узнать, ложным или истинным является утверждение "`myVal` меньше 10", т.е. получить логический результат сравнения.

Для выполнения логических сравнений необходимо использовать логические *операции сравнения*, которые также называются *операциями отношения* и перечислены в табл. 4.1. В этой таблице во всех случаях предполагается, что `var1` является переменной типа `bool`, а типы переменных `var2` и `var3` могут быть всякими.

Такие операции могут применяться в коде с числовыми значениями, например:

```
bool isLessThan10;
isLessThan10 = myVal < 10;
```

Данный фрагмент приведет к присваиванию `isLessThan10` значения `true` в том случае, если в переменной хранится значение меньше `10`, и значения `false` в противном случае. Эти операции могут применяться и в отношении других типов, например, строк:

```
bool isKarli;
isKarli = myString == "Karli";
```

Здесь `isKarli` будет присвоено значение `true` только в том случае, если в переменной `myString` хранится строка `"Karli"`.

Операции сравнения применимы и к логическим значениям, например:

```
bool isTrue;
isTrue = myBool == true;
```

Правда, в этом случае разрешены только операции == и `!=`.

Типичная ошибка в коде часто возникает, когда разработчик ошибочно предполагает, что если `val1 < val2` равно `false`, то `val1 > val2` равно `true`. Однако если `val1 == val2`, то оба этих действия дают `false`.

Таблица 4.1. Операции сравнения

Операция	Категория	Пример	Результат
==	Бинарная	`var1 = var2 == var3;`	`var1` присваивается значение `true`, если `var2` равно `var3`, иначе `var1` присваивается значение `false`
!=	Бинарная	`var1 = var2 != var3;`	`var1` присваивается значение `true`, если `var2` не равно `var3`, иначе `var1` присваивается значение `false`
<	Бинарная	`var1 = var2 < var3;`	`var1` присваивается значение `true`, если `var2` меньше `var3`, иначе `var1` присваивается значение `false`
>	Бинарная	`var1 = var2 > var3;`	`var1` присваивается значение `true`, если `var2` больше `var3`, иначе `var1` присваивается значение `false`
<=	Бинарная	`var1 = var2 <= var3;`	`var1` присваивается значение `true`, если `var2` меньше или равно `var3`, иначе `var1` присваивается значение `false`
>=	Бинарная	`var1 = var2 >= var3;`	`var1` присваивается значение `true`, если `var2` больше или равно `var3`, иначе `var1` присваивается значение `false`

Существует еще несколько других логических операций, которые предназначены специально для работы с булевскими значениями. Они перечислены в табл. 4.2.

Таблица 4.2. Операции для работы с логическими значениями

Операция	Категория	Пример	Результат
!	Унарная	`var1 = !var2;`	`var1` присваивается значение `true`, если `var2` равно `false`, или значение `false`, если `var2` равно `true`. (Логическое НЕ.)
&	Бинарная	`var1 = var2 & var3;`	`var1` присваивается значение `true`, если и `var2`, и `var3` равны `true`; иначе `var1` присваивается значение `false`. (Логическое И.)
|	Бинарная	`var1 = var2 | var3;`	`var1` присваивается значение `true`, если либо `var2`, либо `var3` (либо и `var2`, и `var3`) равна `true`; иначе `var1` присваивается значение `false`. (Логическое ИЛИ.)
^	Бинарная	`var1 = var2 ^ var3;`	`var1` присваивается значение `true`, если либо только `var2`, либо только `var3` (т.е. не и `var2` и `var3` одновременно) равна `true`; иначе `var1` присваивается значение `false`. (Логическое исключающее ИЛИ.)

Тогда приведенный выше фрагмент кода можно представить и так:

```
bool isTrue;
isTrue = myBool & true;
```

Кроме операций & и |, имеются еще две похожие операции, которые называются *условными логическими* операциями и показаны в табл. 4.3.

Таблица 4.3. Условные логические операции

Операция	Категория	Пример	Результат
&&	Бинарная	var1 = var2 && var3;	var1 присваивается значение true, если и var2, и var3 равны true, иначе var1 присваивается значение false. (Логическое И.)
||	Бинарная	var1 = var2 || var3;	var1 присваивается значение true, если либо var2, либо var3 (или var2 и var3) равно true, иначе var1 присваивается значение false. (Логическое ИЛИ.)

Результат этих операций выглядит точно так же, как и у операций & и |, но они отличаются способом получения этого результата и, соответственно, производительностью. Обе они проверяют значение своего первого операнда (var2 в табл. 4.3) и в зависимости от него могут вообще не обрабатывать второй операнд (var3 в табл. 4.3).

Если значение первого операнда операции && равно false, то можно не рассматривать значение второго операнда, поскольку результатом все равно будет false. Точно так же операция || всегда возвращает true, если значение первого операнда равно true, каким бы ни было значение второго операнда. Но это не так для операций & и | — при их использовании всегда вычисляются оба операнда.

Такое условное вычисление операндов в операциях && и || дает несколько более высокую производительность по сравнению с операциями & и |. Это особенно заметно в приложениях, интенсивно использующих такие операции. Поэтому операции && и || рекомендуется применять вместо & и | *везде*, где это возможно. Эти операции действительно очень удобны в более сложных ситуациях, когда вычисление второго операнда возможно только при определенных значениях первого операнда, как, например, в показанном ниже примере:

```
var1 = (var2 != 0) && (var3 / var2 > 2);
```

Здесь если var2 равно 0, деление var3 на var2 приведет либо к возникновению ошибки "деление на ноль", либо к определению переменной var1 как содержащей бесконечное значение (такая ситуация возможна, и ее можно обнаружить в случае некоторых типов, например, float).

Вы можете спросить, а зачем тогда вообще нужны операции & и | ? Причина в том, что эти операции можно применять для выполнения действий с числовыми значениями. На самом деле, как будет показано в разделе "Поразрядные операции", они работают с последовательностями битов, хранящихся в переменных, а не с их значениями.

Булевские операции присваивания

Булевские сравнения можно объединять с присваиваниями с помощью булевских операций присваивания. Они работают точно так же, как и математические операции присваивания (вроде +=, *= и т.д.), о которых рассказывалось в предыдущей главе. Все эти операции перечислены в табл. 4.4.

Таблица 4.4. Булевские операции присваивания

Операция	Категория	Пример выражения	Результат
`&=`	Бинарная	`var1 &= var2;`	`var1` присваивается значение — результат `var1 & var2`
`|=`	Бинарная	`var1 |= var2;`	`var1` присваивается значение — результат `var1 | var2`
`^=`	Бинарная	`var1 ^= var2;`	`var1` присваивается значение — результат `var1 ^ var2`

Эти операции работают с булевскими и числовыми значениям так же, как операции `&`, `|` и `^`.

Учтите, что в присваиваниях `&=` и `|=` используются операции `&` и `|`, а не условные операции `&&` и `||` — т.е. все операнды обрабатываются независимо от значения слева от знака присваивания.

В следующем практическом занятии нужно ввести целое число, и затем выполнить с этим числом различные логические вычисления.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Применение булевских операций

1. Создайте новое консольное приложение с именем `Ch04Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter04`.
2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
  Console.WriteLine("Enter an integer:");
                  // Введите целое число
  int myInt = Convert.ToInt32(Console.ReadLine());
  bool isLessThan10 = myInt < 10;
  bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
  Console.WriteLine("Integer less than 10? {0}", isLessThan10);
                  // Это целое число меньше 10?
  Console.WriteLine("Integer between 0 and 5? {0}", isBetween0And5);
                  // Это целое число находится в диапазоне между 0 and 5?
  Console.WriteLine("Exactly one of the above is true? {0}",
                  // В точности одно из вышеприведенных утверждений верно?
      isLessThan10 ^ isBetween0And5);
  Console.ReadKey();
}
```

*Фрагмент кода **Ch04Ex01\Program.cs***

3. Запустите это приложение, и после появления приглашения введите какое-нибудь целое число. На рис. 4.1 показан результат, который должен получиться.

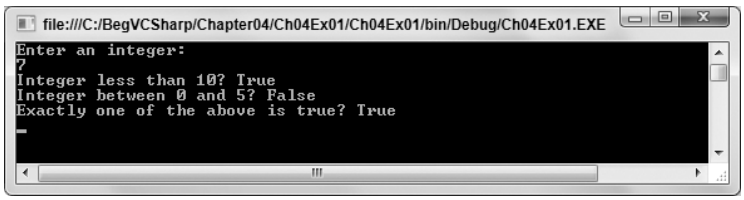


Рис. 4.1. *Результат выполнения приложения* `Ch04Ex01`

Описание работы

В первых двух строках кода выводится приглашение ввести целочисленное значение и выполняется его считывание с помощью уже знакомых операторов:

```
Console.WriteLine("Enter an integer:");
int myInt = Convert.ToInt32(Console.ReadLine());
```

Для получения целого числа из введенной строки используется команда `Convert.ToInt32()` — еще одна команда преобразования из того же семейства, что и уже знакомая нам `Convert.ToDouble()`.

Далее объявляются две логические переменные `isLessThan10` и `isBetween0And5`, которым тут же присваиваются значения в соответствии с их именами:

```
bool isLessThan10 = myInt < 10;
bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
```

Эти переменные используются в трех следующих строках кода: первые две просто выводят значения переменных, а третья выполняет операцию над ними и выводит полученный результат. Проследите работу данного кода в предположении, что пользователь ввел число 7 (см. рис. 4.1).

Первым выводится результат выполнения операции `myInt < 10`. Если `myInt` равно `7`, т.е. меньше `10`, то результатом будет `true`, что и показано на рисунке. Значения `myInt`, равные `10` или больше, дают в результате `false`.

Вторым выводится результат более сложного вычисления `(0 <= myInt) && (myInt <= 5)`. Для этого понадобится выполнить две операции сравнения — ведь нужно выяснить, является ли значение `myInt` большим и равным нулю и меньшим или равным 5 — а затем выполнить над результатами логическую операцию И. В случае значения `7` операция `0 <= myInt` возвращает `true`, а операция `myInt <= 5` — `false`. После этого вычисляется выражение `(true) && (false)`, что дает в итоге значение `false`, что и показано на рисунке.

Последним действием вычисляется логическое исключающее ИЛИ для двух логических переменных `isLessThan10` и `isBetween0And5`. Оно возвращает `true`, только если один из операндов равен `true`, а другой `false` — т.е. если `myInt` равно 6, 7, 8 или 9. Для значения 7, как в нашем примере, результатом будет `true`.

Поразрядные операции

Уже знакомые нам операции `&` и `|` могут выполнять действия и с числовыми значениями. При этом они оперируют с последовательностями битов, хранящихся в переменных, а не со значениями этих переменных, и поэтому их называют *поразрядными* (bitwise) операциями.

В данном разделе мы рассмотрим эти и другие поразрядные операции, определенные в языке C#. За пределами математических приложений такие возможности вряд ли понадобятся.

Рассмотрим вначале операции `&` и `|`. Каждый бит в первом операнде сравнивается с находящимся в такой же позиции битом во втором операнде, и биту результата в этой же позиции присваивается значение из табл. 4.5.

Таблица 4.5. Таблица истинности операции &

Бит в первом операнде	Бит во втором операнде	Бит результата &
1	1	1
1	0	0
0	1	0
0	0	0

В операции | все аналогично, но результирующие биты определяются по-другому, как показано в табл. 4.6.

Таблица 4.6. Таблица истинности операции |

Бит в первом операнде	Бит во втором операнде	Бит результата |
1	1	1
1	0	1
0	1	1
0	0	0

Рассмотрим, к примеру, приведенный ниже фрагмент кода:

```
int result, op1, op2;
op1 = 4;
op2 = 5;
result = op1 & op2;
```

В этом случае нужно взять двоичные представления `op1` и `op2`, т.е. `100` и `101`. Результат получается сравнением двоичных цифр, находящихся в этих двух представлениях в эквивалентных позициях.

- Если крайний левый бит и в `op1`, и в `op2` равен `1`, то крайний левый бит в `result` тоже будет равен `1`, а если нет, то он будет равен `0`.
- Если следующий бит и в `op1`, и в `op2` равен `1`, то следующий бит в `result` тоже будет равен `1`, а если нет, то он будет равен `0`.
- Аналогично определяются и все остальные биты.

В данном примере крайние левые биты у `op1` и `op2` равны 1, поэтому у `result` крайний левый бит тоже равен 1. Вторые биты у обоих равны 0, а третьи — 1 и 0, поэтому второй и третий биты у `result` будут равны 0. Следовательно, конечное значение `result` в двоичном представлении равно `100`, т.е. `result` получает значение `4`. Весь этот процесс графически представлен ниже.

```
  1 0 0        4
& 1 0 1      & 5
-------      ---
  1 0 0        4
```

Аналогично выполняется и операция |, но только результирующий бит будет равен 1, если хотя бы один из битов в этой же позиции в операндах равен 1:

```
  1 0 0        4
| 1 0 1      | 5
-------      ---
  1 0 1        5
```

Подобным образом можно использовать и операцию ^, когда результирующий бит равен 1, если какой-то один бит (но не оба сразу), находящийся в той же позиции в операндах, равен 1, как показано в табл. 4.7.

Таблица 4.7. Таблица истинности операции ^

Бит в первом операнде	Бит во втором операнде	Бит результата ^
1	1	0
1	0	1
0	1	1
0	0	0

В языке C# имеется также унарная поразрядная операция ~, которая инвертирует каждый бит операнда — тогда переменная-результат содержит значения 1 в тех битах, которые в операнде равны 0, и значения 0 в тех битах, которые в операнде равны 1 (табл. 4.8).

Таблица 4.8. Таблица истинности операции ~

Бит в операнде	Бит результата ~
1	0
0	1

Способ, которым числа типа `int` сохраняются в .NET, называется *дополнительным двоичным кодом*. Это означает, что иногда результаты унарной операции ~ могут выглядеть несколько странно. Поскольку, к примеру, тип `int` представляет собой состоящее из 32 битов число, то знание того, каким образом операция ~ действует на все эти 32 бита, может помочь понять, что же происходит. Например, число 5 в своем полном двоичном представлении выглядит следующим образом:

```
00000000000000000000000000000101
```

А число –5 — так:

```
11111111111111111111111111111011
```

Дело в том, что в дополнительном двоичном коде значение `-x` определяется как `~x + 1`. Это может показаться странным, но такая система очень удобна для сложения чисел. Например, сложение 10 и -5 (т.е. вычитание 5 из 10) в двоичном формате выглядит следующим образом:

```
  00000000000000000000000000001010
+ 11111111111111111111111111111011
= 100000000000000000000000000000101
```

Если не обращать внимания на самую левую 1, то остается двоичное представление числа 5. Так что результаты вроде ~1 = −2 могут выглядеть странно, но таковы уж структуры их представления.

Поразрядные операции, продемонстрированные в этом разделе, весьма полезны в определенных ситуациях, поскольку позволяют легко использовать для хранения информации отдельные биты переменной. Рассмотрим простой пример представления цвета с применением трех битов для указания содержимого `red`, `green` и `blue`. Эти биты можно задавать независимо друг от друга, определяя трехбитные сочетания, которые показаны в табл. 4.9.

Таблица 4.9. Использование битов для хранения цветов

Биты	Десятичное представление	Значение
000	0	Черный (`black`)
100	4	Красный (`red`)
010	2	Зеленый (`green`)
001	1	Голубой (`blue`)
101	5	Пурпурный (`magenta`)
110	6	Желтый (`yellow`)
011	3	Бирюзовый (`cyan`)
111	7	Белый (`white`)

Предположим, что все эти значения хранятся в переменной типа `int`. Тогда, начав с черного цвета, т.е. со значения 0, можно выполнить следующие операции:

```
int myColor = 0;
bool containsRed;
myColor = myColor | 2;              // Установка зеленого бита -
                                    // теперь myColor содержит 010
myColor = myColor | 4;              // Установка красного бита -
                                    // теперь myColor содержит 110
containsRed = (myColor & 4) == 4;   // Проверка значения красного бита
```

В последней строке приведенного кода `containsRed` будет присвоено значение `true`, поскольку бит, представляющий красный цвет в `myColor`, действительно равен 1. Такой прием может оказаться весьма полезным для эффективного использования информации, поскольку поразрядные операции позволяют проверить значения сразу всех битов (32 битов в случае значений типа `int`). Однако существуют и более удобные способы хранения дополнительной информации в одиночных переменных (с помощью более сложных типов переменных, о которых будет рассказано в следующей главе).

Помимо этих четырех поразрядных операций, мы рассмотрим в настоящем разделе еще две, которые перечислены в табл. 4.10.

Таблица 4.10. Поразрядные операции >> и <<

Операция	Категория	Пример выражения	Результат
`>>`	Бинарная	`var1 = var2 >> var3;`	`var1` присваивается значение, получаемое сдвигом вправо двоичного содержимого `var2` на указанное в `var3` количество разрядов
`<<`	Бинарная	`var1 = var2 << var3;`	`var1` присваивается значение, получаемое сдвигом влево двоичного содержимого `var2` на указанное в `var3` количество разрядов

Эти операции, которые обычно называют *поразрядными операциями сдвига*, лучше всего продемонстрировать на коротком примере:

```
int var1, var2 = 10, var3 = 2;
var1 = var2 << var3;
```

Здесь переменной var1 присвоено значение 40: в двоичном представлении число 10 выглядит как 1010, а после сдвига на две позиции влево оно превращается в 101000, что является двоичным представлением числа 40. По сути, выполняется операция умножения. Каждый сдвиг влево на один бит эквивалентен умножению на 2, поэтому сдвиг на два бита влево эквивалентен умножению на 4. А сдвиг на один бит вправо эквивалентен делению операнда на 2 с отбрасыванием возможного остатка:

```
int var1, var2 = 10;
var1 = var2 >> 1;
```

В данном примере в переменной var1 образуется значение 5, а после выполнения следующего кода результат будет равен 2:

```
int var1, var2 = 10;
var1 = var2 >> 2;
```

Эти операции бывают нужны нечасто, но знать об их существовании не помешает. Их главной сферой применения является высоко оптимизированный код, в котором расходы на выполнение других математических операций просто недопустимы. Из-за этого они часто применяются, к примеру, в драйверах устройств или в системном коде.

У поразрядных операций сдвига тоже имеются операции присваивания, которые перечислены в табл. 4.11.

Таблица 4.11. Операции присваивания для поразрядных операций сдвига

Операция	Категория	Пример выражения	Результат
>>=	Унарная	var1 >>= var2;	var1 присваивается значение, получаемое сдвигом вправо двоичного содержимого var1 на указанное в var2 количество разрядов
<<=	Унарная	var1 <<= var2;	var1 присваивается значение, получаемое сдвигом влево двоичного содержимого var1 на указанное в var2 количество разрядов

Более полная таблица приоритетов операций

Теперь, когда были рассмотрены еще несколько операций, пора дополнить приведенную в предыдущей главе таблицу приоритетов операций. Сейчас она выглядит так, как показано в табл. 4.12.

Таблица 4.12. Дополненная таблица приоритетов операций

Приоритет	Операции
Высший	++, -- (префиксные); (), +, - (унарная); !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&

Окончание табл. 4.12

Приоритет	Операции
	^
	|
	&&
	||
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
Низший	++, -- (постфиксные)

Эта таблица содержит немало новых уровней, зато она четко определяет, каким образом будут вычисляться выражения вроде нижеприведенного, где операция `&&` выполняется после операций <= и >=:

```
var1 = var2 <= 4 && var2 >= 2;
```

Для большей ясности не стесняйтесь добавлять в подобные выражения круглые скобки. Компилятор знает порядок выполнения операций, а вот люди часто его забывают (или бывает просто нужно изменить порядок вычисления выражения). Явное указание порядка вычислений в вышеприведенном выражении устранит эту проблему:

```
var1 = (var2 <= 4) && (var2 >= 2);
```

Оператор `goto`

Язык C# позволяет снабжать строки кода метками и переходить к ним с помощью оператора `goto`. У такого подхода имеются как преимущества, так и недостатки. Основное преимущество — это простой способ для указания, какой код когда должен выполняться, а главный недостаток — чрезмерное применение оператора `goto` может сделать код запутанным и трудным для понимания.

Оператор `goto` используется следующим образом:

```
goto <имяМетки>;
```

Метки определяются следующим образом:

```
<имяМетки>:
```

Например, рассмотрим такой код:

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

Выполнение этого кода происходит так:

- Сначала объявляется переменная `myInteger` с типом `int`, которой тут же присваивается значение `5`.
- Далее оператор `goto` прерывает обычный ход выполнения кода и передает управление строке, помеченной `myLabel:`.
- После этого в окно консоли выводится значение `myInteger`.

Получается, что выделенная строка не будет выполняться:

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

Вообще-то при попытке компиляции данного кода в окне Error List (Список ошибок) появится предупреждение с заголовком "Unreachable code detected" (Обнаружен недостижимый код) и информацией о местоположении проблемного кода. Кроме того, под `myInteger` в недостижимой строке кода появится волнистая зеленая линия.

Операторы `goto` имеют право на существование, но они могут сильно запутать код. Вообще-то вполне можно обойтись и без них (и с помощью приемов, описанных ниже в данной главе, вы сможете это делать). Ниже показан пример такого запутанного кода:

```
start:
int myInteger = 5;
goto addVal;
writeResult:
Console.WriteLine("myInteger = {0}", myInteger);
goto start;
addVal:
myInteger += 10;
goto writeResult;
```

Этот код вполне допустим, но очень труден для чтения. Попробуйте прочитать его сами и убедитесь в этом. Но сначала попытайтесь, глядя на код, определить, что он будет делать, и если догадаетесь, можете похвалить себя. Мы еще снова вернемся к оператору `goto` позже в этой главе, в связи с особенностями его использования с некоторыми другими структурами.

Ветвление

Ветвлением называется управление тем, какая строка кода должна выполняться следующей. Строку, на которую должен осуществляться переход, определяет один из видов условного оператора. Действие этого условного оператора основано на сравнении проверочного значения с одним или более возможными значениями с применением булевской логики.

В настоящем разделе описаны три имеющихся в C# способа ветвления:

- тернарная операция;
- оператор `if`;
- оператор `switch`.

Тернарная операция

Самый простой способ выполнить сравнение — это воспользоваться *тернарной* (или *условной*) операцией, о которой упоминалось в предыдущей главе. Унарные операции, работающие с одним операндом, уже встречались, бинарные операции, работающие с двумя операндами — тоже, поэтому не должен вызывать удивление тот факт, что существует операция, работающая и с тремя операндами. Синтаксис этой операции выглядит следующим образом:

```
<проверка> ? <результатЕслиTrue> : <результатЕслиFalse>
```

Здесь *<проверка>* вычисляется для получения логического значения, а результатом операции в зависимости от этого значения является либо *<результатЕслиTrue>*, либо *<результатЕслиFalse>*.

Это позволяет, например, проверить значение целой переменной с именем myInteger:

```
string resultString = (myInteger < 10) ? "Меньше 10" : "Больше или равно 10";
```

Результатом приведенной тернарной операции будет какая-то одна из двух строк, каждая из которых может быть присвоена переменной resultString. Выбор того, какая из них будет присвоена, выполняется сравнением значения myInteger с числом 10: если это значение меньше десяти, будет присвоена первая строка, а если больше или равно 10, то вторая. Например, если значение myInteger равно 4, то resultString будет содержать строку Меньше 10.

Такая операция вполне подходит для простых присваиваний, подобных показанному, но неудобна для выполнения более длинных фрагментов кода на основании сравнения. Гораздо лучше в таких случаях использовать оператор if.

Оператор if

Оператор if является гораздо более гибким и удобным средством для принятия решений. В отличие от выражений ?:, оператор if не имеет результата (поэтому его нельзя использовать в операциях присваивания); зато он пригоден для условного выполнения других операторов.

Наиболее простой способ использования оператора if выглядит так, как показано ниже: вычисляется значение выражения <*проверка*>, и следующая за ним строка кода выполняется, только если <*проверка*> возвращает true:

```
if (<проверка>)
   <код, выполняемый, если <проверка> дает true>;
```

После выполнения этого кода — или невыполнения из-за того, что результат вычисления выражения <*проверка*> равен false — выполнение программы продолжается со следующей строки кода.

Вместе с оператором if можно указать и дополнительный код с помощью оператора else. Этот оператор выполняется в случае, если при вычислении выражения <*проверка*> получается false:

```
if (<проверка>)
   <код, выполняемый, если <проверка> дает true>;
else
   <код, выполняемый, если <проверка> дает false>;
```

Оба раздела кода могут содержать несколько операторов, если оформить их как заключенные в фигурные скобки блоки:

```
if (<проверка>)
{
   <код, выполняемый, если <проверка> дает true>;
}
else
{
   <код, выполняемый, если <проверка> дает false>;
}
```

В качестве простого примера можно переписать приведенный в предыдущем разделе код с тернарной операцией:

```
string resultString = (myInteger < 10) ? "Меньше 10" : "Больше или равно 10";
```

Поскольку результат оператора if не может присваиваться переменной, потребуется отдельный шаг:

```
string resultString;
if (myInteger < 10)
  resultString = "Меньше 10";
else
  resultString = "Больше или равно 10";
```

Такой код более многословен, но зато его значительно легче читать и понимать, чем эквивалентную тернарную форму; кроме того, он обеспечивает гораздо бо́льшую гибкость.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Применение оператора `if`

1. Создайте новое консольное приложение с именем `Ch04Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter04`.

2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
  string comparison;
  Console.WriteLine("Enter a number:");          // Введите число
  double var1 = Convert.ToDouble(Console.ReadLine());
  Console.WriteLine("Enter another number:");    // Введите еще одно число
  double var2 = Convert.ToDouble(Console.ReadLine());
  if (var1 < var2)
    comparison = "less than";                    // меньше, чем
  else
  {
    if (var1 == var2)
      comparison = "equal to";                   // такое же, как
    else
      comparison = "greater than";               // больше, чем
  }
  Console.WriteLine("The first number is {0} the second number.", comparison);
                  // Первое число ... второе число
  Console.ReadKey();
}
```

Фрагмент кода ***Ch04Ex02\Program.cs***

3. Запустите приложение и в ответ на приглашения введите два числа, как показано на рис. 4.2.

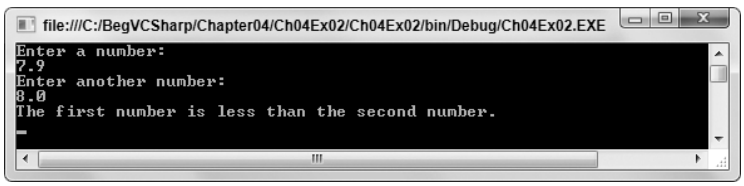


Рис. 4.2. *Приложение* `Ch04Ex02` *в действии*

Описание работы

Первый раздел приведенного кода выглядит очень знакомо. Он просто получает от пользователя два значения типа `double`:

```
string comparison;
Console.WriteLine("Enter a number:");
double var1 = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Enter another number:");
double var2 = Convert.ToDouble(Console.ReadLine());
```

Далее на основании значений, полученных в var1 и var2, переменной comparison типа string присваивается некоторая строка. Сначала выполняется проверка, меньше ли значение var1 значения var2:

```
if (var1 < var2)
  comparison = "less than";
```

Если нет, то значение var1 либо больше, либо равно значению var2. Поэтому внутри раздела else первой операции сравнения нужна еще одна, вложенная, операция сравнения:

```
else
{
  if (var1 == var2)
    comparison = "equal to";
```

Раздел else второй операции сравнения выполняется, только если значение var1 больше значения var2:

```
  else
    comparison = "greater than";
}
```

И, наконец, значение переменной value выводится на консоль:

```
Console.WriteLine("The first number is {0} the second number.", comparison);
```

Примененное здесь вложение операторов if является лишь одним из возможных способов. Аналогичное сравнение можно было бы записать и так:

```
if (var1 < var2)
  comparison = "less than";
if (var1 == var2)
  comparison = "equal to";
if (var1 > var2)
  comparison = "greater than";
```

Недостатком такого подхода является то, что в нем выполняются все три операции сравнения, независимо от значений var1 и var2. А в первом подходе выполняется только одна операция сравнения, если var1 меньше var2, и две в противном случае (вместе с операцией сравнения var1 == var2), что сокращает количество выполняемых строк кода. Здесь разница в показателях производительности незначительна, однако в тех приложениях, где скорость выполнения важна, она может играть решающую роль.

Проверка нескольких условий с помощью операторов `if`

В предыдущем примере выполнялась проверка трех условий с переменной var1, что охватывало все возможные ее значения. Однако иногда может понадобиться проверить переменную на соответствие каким-то конкретным значениям, например, равна ли var1 числу 1, 2, 3 или 4, и т.д. В таких случаях описанный в предыдущем разделе подход приводит к многоуровневому вложению кода:

```
if (var1 == 1)
{
  // Выполнение какого-нибудь действия.
}
else
{
  if (var1 == 2)
  {
    // Выполнение какого-то другого действия.
  }
```

```
    else
    {
      if (var1 == 3 || var1 == 4)
      {
        // Выполнение еще какого-то действия.
      }
      else
      {
        // Выполнение еще какого-нибудь другого действия.
      }
    }
}
```

Одна из типичных ошибок – запись условий вроде `if (var1 == 3 || var1 == 4)` в виде `if (var1 == 3 || 4)`. Здесь, согласно приоритетам, первой будет выполнена операция ==, и операции || достанутся булевский и числовой операнд, что приведет к ошибке.

В таких ситуациях лучше применять нескольку иную схему отступов и сжимать блоки `else` (т.е. использовать после `else` одну строку кода, а не блок). Такой подход приводит к структуре с операторами `else if`:

```
if (var1 == 1)
{
  // Выполнение какого-нибудь действия.
}
else if (var1 == 2)
{
  // Выполнение какого-то другого действия.
}
else if (var1 == 3 || var1 == 4)
{
  // Выполнение еще какого-то действия.
}
else
{
  // Выполнение еще какого-нибудь другого действия.
}
```

Эти операторы `else if` на самом деле являются двумя отдельными операторами, а код по функциональности полностью идентичен предыдущему, но гораздо более удобен для чтения. При выполнении нескольких сравнений подряд, как в этом примере, в качестве альтернативной структуры для выполнения ветвления можно использовать оператор `switch`.

Оператор `switch`

Оператор `switch` похож на `if` тем, что он тоже условно выполняет код на основании проверяемого значения. Однако в отличие от него, `switch` позволяет сразу проверять переменную на равенство нескольким значениям, а не выполнять проверки различных условий. В таких проверках разрешены только сравнения с дискретными значениями, а не конструкции вроде “больше чем X”, поэтому и способ применения этого оператора немного отличается.

Базовая структура оператор `switch` выглядит следующим образом:

```
switch (<проверяемаяПеременная>)
{
  case <значениеДляСравнения1>:
```

```
    <код, выполняемый, если <проверяемаяПеременная> == <значениеДляСравнения1> >
    break;
  case <значениеДляСравнения2>:
    <код, выполняемый, если <проверяемаяПеременная> == <значениеДляСравнения2> >
    break;
  ...
  case <значениеДляСравненияN>:
    <код, выполняемый, если <проверяемаяПеременная> == <значениеДляСравненияN> >
    break;
  default:
    <код, выполняемый, если <проверяемаяПеременная> != <значениямДляСравнения> >
    break;
}
```

Значение выражения <*проверяемаяПеременная*> сравнивается с каждым из значений <*значениеДляСравненияX*> (задаваемых в операторах `case`), и если обнаруживается совпадение, то выполняется код из раздела, соответствующего обнаруженному совпадению. Если не обнаружено ни одного совпадения, выполняется код, который содержится в разделе `default` — если такой раздел существует.

В конце кода в каждом разделе обычно добавляется дополнительная команда `break`, исключающая переход после обработки одного блока `case` к обработке следующего оператора `case`.

Это поведение — одно из отличий языка C# от C++, в котором разрешено переходить от обработки одного оператора `case` к другому.

В данном случае оператор `break` просто завершает работу оператора `switch`, после чего выполнение продолжается с оператора, следующего после данной структуры.

Существуют и другие способы для предотвращения перехода потока выполнения в коде C# от одного оператора `case` к следующему. Можно использовать оператор `return`, который завершает выполнение текущей функции, а не только структуры `switch` (о нем будет рассказываться в главе 6), или оператор `goto`. Операторы `goto` (которые рассматривались ранее в главе) пригодны для подобных случаев, поскольку операторы `case` фактически определяют метки в коде. Например:

```
switch (<проверяемаяПеременная>)
{
  case <значениеДляСравнения1>:
    <код, выполняемый, если <проверяемаяПеременная> == <значениеДляСравнения1> >
    goto case <значениеДляСравнения2>;
  case <значениеДляСравнения2>:
    <код, выполняемый, если <проверяемаяПеременная> == <значениеДляСравнения2> >
    break;
  ...
```

У правила, что выполнение одного оператора `case` не может свободно переходить на следующий, имеется одно исключение: при записи нескольких операторов `case` подряд перед одним блоком кода фактически выполняется проверка сразу нескольких условий. При удовлетворении любого из этих условий код будет выполнен. Например:

```
switch (<проверяемая_переменная>)
{
  case <значениеДляСравнения1>:
  case <значениеДляСравнения2>:
   <код, выполняемый, если <проверяемаяПеременная> == <значениеДляСравнения1> или
                           <проверяемаяПеременная> == <значениеДляСравнения2> >
    break;
  ...
```

Объединять с операторами `case` можно и оператор `default`. Этот оператор не обязательно должен быть последним в списке сравнений, поэтому при желании его можно помещать в одну последовательность с операторами `case`. Добавление точки прерывания с помощью оператора `break`, `goto` или `return` обеспечивает существование правильного пути выполнения кода в структуре во всех случаях.

Каждая из конструкций *<значениеДляСравненияX>* должна представлять собой константное значение. Один из способов сделать так — использование литеральных значений:

```
switch (myInteger)
{
  case 1:
    <код, выполняемый, если myInteger == 1>
    break;
  case -1:
    <код, выполняемый, если myInteger == -1>
    break;
  default:
    <код, выполняемый, если myInteger != НиОдномуЗначению>
    break;
}
```

Второй способ — применение *константных переменных*. Константные переменные (для простоты называемые просто *константами*) похожи на любые другие переменные во всем, кроме одного: их значение никогда не изменяется. После присваивания константной переменной значения оно не меняется на протяжении всего выполнения кода. Константные переменные часто бывают очень удобны, поскольку код, в котором фактические сравниваемые значения заменены описательными именами, читать обычно легче.

Объявляются константные переменные путем указания перед идентификатором типа ключевого слова `const`, а также *обязательного* присваивания им значения:

```
const int intTwo = 2;
```

Данный код вполне допустим, но вот такой:

```
const int intTwo;
intTwo = 2;
```

приведет к появлению ошибки и невозможности его компиляции. То же самое произойдет и при попытке изменить значение константной переменной любым другим способом после присваивания ей первоначального значения.

В следующем практическом занятии демонстрируется применение оператора `switch` для вывода в окне консоли различных строк в зависимости от введенного пользователем значения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Применение оператора `switch`

1. Создайте новое консольное приложение с именем `Ch04Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter04`.

2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
  const string myName = "karli";
  const string sexyName = "angelina";
  const string sillyName = "ploppy";
  string name;
  Console.WriteLine("What is your name?");
                  // Как вас зовут?
  name = Console.ReadLine();
```

```
      switch (name.ToLower())
      {
         case myName:
            Console.WriteLine("You have the same name as me!");
                           // У вас такое же имя, как и у меня!
            break;
         case sexyName:
            Console.WriteLine("My, what a sexy name you have!");
                           // Чудесное у вас имя!
            break;
         case sillyName:
            Console.WriteLine("That's a very silly name.");
                           // Это очень глупое имя.
            break;
      }
      Console.WriteLine("Hello {0}!", name);
                        // Привет
      Console.ReadKey();
   }
```

Фрагмент кода ***Ch04Ex03\Program.cs***

3. Запустите это приложение и в ответ на приглашение введите какое-нибудь имя (рис. 4.3).

Рис. 4.3. *Приложение* `Ch04Ex03` *в действии*

Описание работы

В приведенном коде создаются три константных переменных строкового типа, считывается введенная пользователем строка, и на ее основании на консоль выводится некоторый текст. Здесь пользователь должен вводить имена.

Перед сравнением введенного пользователем имени (в переменной `name`) с константными переменными оно сначала преобразуется в нижний регистр с помощью метода `name.ToLower()`. Эта стандартная команда работает со всеми строковыми переменными и очень удобна, когда нет уверенности в том, каким образом пользователь ввел имя. После ее применения имена `Karli`, `kArLi`, `karli` и т.д. все равно будут совпадать с проверочной строкой `karli`.

Сам оператор `switch` сравнивает введенную строку со значениями определенных константных переменных, и при успешном сравнении выполняется вывод на консоль персонализированного приветствия пользователю, а иначе на консоль выводится универсальное общее приветствие.

Операторы `switch` не имеют ограничения на количество вложенных разделов `case`, поэтому в приведенный код можно добавить и другие возможные имена, хотя это, конечно, потребует времени.

Циклы

Циклическое выполнение означает многократное выполнение операторов. Это наиболее мощная возможность во всем программировании, поскольку она позволяет выполнять

необходимые операции столько раз, сколько требуется, без повторного написания одного и того же кода.

В качестве простого примера рассмотрим следующий код, вычисляющий сумму денег на банковском счету через 10 лет при условии ежегодной выплаты процентов без снятий и поступлений средств:

```
double balance = 1000;
double interestRate = 1.05; // годовая ставка 5%
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

Десятикратная запись одной и той же строки сама по себе неэффективна, а если понадобится заменить срок 10 лет каким-нибудь другим значением? Тогда придется вручную копировать эту строку кода требуемое число раз, что весьма утомительно. К счастью, это совсем необязательно. Вместо этого можно создать цикл, выполняющий необходимую инструкцию нужное количество раз.

Имеется еще один важный вид циклов, выполняющийся до тех пор, пока не будет удовлетворено определенное условие. Такие циклы выглядят немного проще, чем вышеописанные (хотя и не менее полезны), поэтому давайте с них и начнем.

Циклы do

Циклы `do` действуют следующим образом. Сначала выполняется код тела цикла, затем производится проверка логического условия, и если она возвращает `true`, то тело цикла выполняется снова — и так до тех пор, пока проверка не возвратит `false`, после чего цикл завершается. Ниже приведен синтаксис цикла `do`; подразумевается, что выражение *<проверка>* должно возвращать одно из булевских значений:

```
do
{
   <код тела цикла>
} while (<проверка>);
```

Символ точки с запятой после оператора `while` обязателен.

Например, для вывода чисел от 1 до 10 в столбик можно использовать такой цикл `do`:

```
int i = 1;
do
{
   Console.WriteLine("{0}", i++);
} while (i <= 10);
```

Здесь постфиксная версия операции ++ увеличивает значение `i` на 1 после его вывода на экран, поэтому проверка должна иметь вид `i <= 10`, чтобы число 10 также было выведено на консоль.

В следующем практическом занятии похожий код выводит результат вычисления баланса на счету по прошествии количества лет, необходимого для накопления на этом счету указанной суммы денег, исходя из заданной начальной суммы и фиксированной процентной ставки.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Применение циклов do

1. Создайте новое консольное приложение с именем `Ch04Ex04` и сохраните его в каталоге `C:\BegVCSharp\Chapter04`.

2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
  double balance, interestRate, targetBalance;
  Console.WriteLine("What is your current balance?");
                  // Каков текущий баланс?
  balance = Convert.ToDouble(Console.ReadLine());
  Console.WriteLine("What is your current annual interest rate (in %)?");
                  // Какова ежегодная ставка (в процентах)?
  interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
  Console.WriteLine("What balance would you like to have?");
                  // Какой баланс необходимо получить?
  targetBalance = Convert.ToDouble(Console.ReadLine());
  int totalYears = 0;
  do
  {
    balance *= interestRate;
    ++totalYears;
  }
  while (balance < targetBalance);
  Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                totalYears, totalYears == 1 ? "" : "s", balance);
                  // Вывод баланса через вычисленное количество лет
  Console.ReadKey();
}
```

Фрагмент кода ***Ch04Ex04\Program.cs***

3. Запустите это приложение и в ответ на приглашение введите какие-нибудь значения (рис. 4.4).

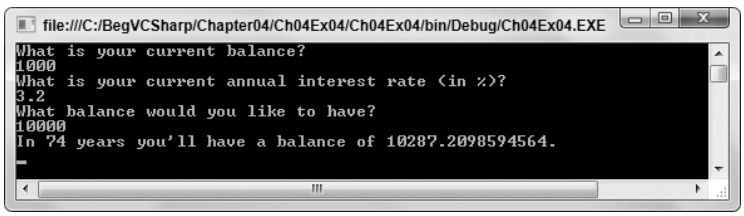


Рис. 4.4. *Приложение* `Ch04Ex04` *в действии*

Описание работы

В приведенном коде операция ежегодного подсчета баланса с фиксированной процентной ставкой просто повторяется такое количество раз, которое необходимо, чтобы баланс стал удовлетворять конечному условию. Подсчет количества необходимых для этого лет ведется путем увеличения на 1 значения переменной-счетчика в каждой итерации цикла:

```
int totalYears = 0;
do
{
  balance *= interestRate;
  ++totalYears;
}
while (balance < targetBalance);
```

После этого значение данной переменной-счетчика можно использовать как часть выводимого на экран результата:

```
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                  totalYears, totalYears == 1 ? "" : "s", balance);
```

Такое применение тернарной операции `?:` для условного форматирования текста с минимумом кода является, пожалуй, наиболее распространенным подходом. В данном случае эта операция используется для вывода после слова `year` окончания множественного числа `s` в случае, если значение `totalYears` не равно `1`.

К сожалению, данный код далеко не идеален. Например, если пользователь укажет желаемый баланс меньше, чем текущий баланс, то вывод будет выглядеть так, как показано на рис. 4.5.

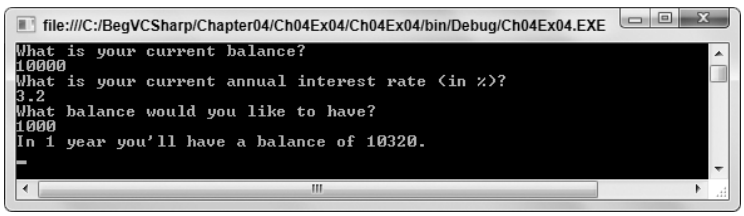


Рис. 4.5. *Вывод приложения `Ch04Ex04` в случае, если указанный желаемый баланс меньше текущего*

Циклы `do` всегда выполняются как минимум один раз. Иногда, как, например, в данной ситуации, такой вариант нежелателен. Разумеется, можно добавить оператор `if`:

```
int totalYears = 0;
if (balance < targetBalance)
{
  do
  {
    balance *= interestRate;
    ++totalYears;
  }
  while (balance < targetBalance);
}
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                  totalYears, totalYears == 1 ? "" : "s", balance);
```

Очевидно, что это все-таки усложняет код. Поэтому гораздо лучшим решением является цикл `while`.

Циклы `while`

Циклы `while` очень похожи на циклы `do`, но имеют одно важное отличие: логическая проверка в них выполняется в начале, а не в конце цикла. Если проверка сразу возвращает `false`, код тела цикла вообще не выполняется, а поток выполнения переходит на код, следующий за циклом.

Синтаксис цикла `while` выглядит следующим образом:

```
while (<проверка>)
{
  <код тела цикла>
}
```

Такие циклы можно применять практически так же, как циклы do:

```
int i = 1;
while (i <= 10)
{
  Console.WriteLine("{0}", i++);
}
```

Этот код делает то же самое, что и приведенный ранее код с циклом do — он выводит в столбик числа от 1 до 10. В следующем практическом занятии показано, как подправить приложение из предыдущего раздела с помощью цикла while.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Применение циклов `while`

1. Создайте новое консольное приложение с именем Ch04Ex05 и сохраните его в каталоге C:\BegVCSharp\Chapter04.
2. Добавьте в файл Program.cs следующий код (используйте код из приложения Ch04Ex04, только не забудьте удалить оператор while в конце цикла):

```
static void Main(string[] args)
{
  double balance, interestRate, targetBalance;
  Console.WriteLine("What is your current balance?");
  balance = Convert.ToDouble(Console.ReadLine());
  Console.WriteLine("What is your current annual interest rate (in %)?");
  interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
  Console.WriteLine("What balance would you like to have?");
  targetBalance = Convert.ToDouble(Console.ReadLine());
  int totalYears = 0;
  while (balance < targetBalance)
  {
    balance *= interestRate;
    ++totalYears;
  }
  Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                totalYears, totalYears == 1 ? "" : "s", balance);
  if(totalYears==0) Console.WriteLine(
    "To be honest, you really didn't need to use this calculator.");
   // Ну и зачем вам нужен этот калькулятор?
  Console.ReadKey();
}
```

Фрагмент кода ***Ch04Ex05\Program.cs***

3. Запустите приложение, но на этот раз укажите в качестве желаемого баланс, меньший текущего (рис. 4.6).

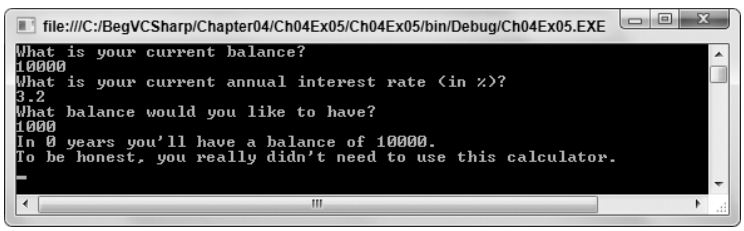


Рис. 4.6. *Приложение* Ch04Ex05 *в действии*

Описание работы

Простая замена цикла do циклом while устранила проблему из предыдущего примера. Перемещение логической проверки в начало обеспечивает правильность вычисления в случае, когда выполнение цикла не нужно, и переходит сразу к выводу результата.

Конечно, возможны и другие варианты обработки такой ситуации. Например, после ввода данных можно проверить, больше ли указанный желаемый баланс, чем текущий. Тогда лучше поместить раздел ввода пользователем данных в цикл:

```
Console.WriteLine("What balance would you like to have?");
do
{
  targetBalance = Convert.ToDouble(Console.ReadLine());
  if (targetBalance <= balance)
    Console.WriteLine("You must enter an amount greater than " +
      "your current balance!\nPlease enter another value.");
    // Желаемый баланс должен быть больше текущего! Введите другое значение.
}
while (targetBalance <= balance);
```

Теперь все бессмысленные значения будут отбрасываться, как показано на рис. 4.7.

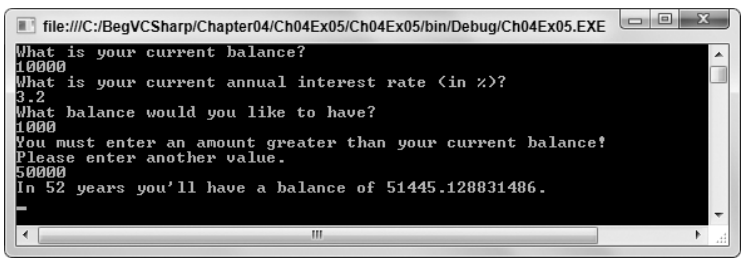


Рис. 4.7. *Вывод усовершенствованного приложения* `Ch04Ex05`

Подобная *проверка достоверности*, или *верификация*, вводимых пользователем данных довольно важна при проектировании приложения, и в настоящей книге будет встречаться немало примеров ее применения.

Циклы `for`

Последним видом цикла, который осталось рассмотреть в главе, является for. Цикл этого типа выполняется определенное количество раз и использует собственный счетчик. Для определения цикла for требуется следующая информация:

- начальное значение для инициализации переменной-счетчика;
- условие для продолжения цикла, в котором участвует переменная-счетчик;
- действие с переменной-счетчиком в конце каждого цикла.

Например, можно написать такой цикл for, чтобы значение его счетчика увеличивалось на единицу с 1 до 10. Тогда в качестве начального значения потребуется указать 1, в качестве условия — то, что значение счетчика должно быть меньше или равно 10, а в качестве действия в конце каждой итерации цикла — увеличение счетчика на 1.

Эта информация указывается в структуре цикла loop следующим образом:

```
for (<инициализация>; <условие>; <действие>)
{
  <код тела цикла>
}
```

Этот цикл работает точно так же, как и приведенный ниже цикл `while`:

```
<инициализация>
while (<условие>)
{
   <код тела цикла>
   <действие>
}
```

Формат цикла `for` просто делает код более понятным, поскольку его синтаксис обеспечивает указание всех деталей цикла в одном месте, вместо разнесения их по нескольким операторам в разных частях кода.

Ранее мы уже выводили числа от 1 до 10 в столбик с помощью циклов `do` и `while`. Ниже приведен код, демонстрирующий, как то же самое можно сделать с помощью цикла `for`:

```
int i;
for (i = 1; i <= 10; ++i)
{
   Console.WriteLine("{0}", i);
}
```

Здесь счетчик представляет собой переменную типа `int` с именем `i` и начальным значением `1`, которое увеличивается на 1 в конце каждой итерации цикла. В теле цикла значение `i` выводится на консоль.

После выхода потока управления из цикла переменная `i` содержит значение `11`. После последней итерации цикла, в которой значение переменной `i` равно 10, оно еще увеличивается до 11, поскольку это происходит до проверки условия `i <= 10`, после которой цикл завершается. Как и циклы `while`, цикл `for` выполняется только в том случае, если перед началом первой итерации проверка условия возвращает `true` — а это означает, что тело цикла может вообще не выполниться.

И еще один важный момент: переменную-счетчик можно объявить и внутри оператора `for`, т.е. приведенный выше код можно записать и так:

```
for (int i = 1; i <= 10; ++i)
{
   Console.WriteLine("{0}", i);
}
```

Но тогда к переменной `i` невозможно будет обратиться из кода за пределами цикла (об этом будет рассказано в главе 6).

В следующем практическом занятии будут использоваться циклы `for`, а т.к. с концепцией циклов мы уже знакомы, этот пример будет более интересным: отображение множества Мандельброта (только с помощью обычных текстовых символов, так что это будет не так красиво).

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Применение циклов `for`

1. Создайте новое консольное приложение с именем `Ch04Ex06` и сохраните его в каталоге `C:\BegVCSharp\Chapter04`.

2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
   double realCoord, imagCoord;
   double realTemp, imagTemp, realTemp2, arg;
   int iterations;
   for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
   {
```

```
      for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
      {
        iterations = 0;
        realTemp = realCoord;
        imagTemp = imagCoord;
        arg = (realCoord * realCoord) + (imagCoord * imagCoord);
        while ((arg < 4) && (iterations < 40))
        {
          realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
          imagTemp = (2 * realTemp * imagTemp) - imagCoord;
          realTemp = realTemp2;
          arg = (realTemp * realTemp) + (imagTemp * imagTemp);
          iterations += 1;
        }
        switch (iterations % 4)
        {
          case 0:
            Console.Write(".");
            break;
          case 1:
            Console.Write("o");
            break;
          case 2:
            Console.Write("O");
            break;
          case 3:
            Console.Write("@");
            break;
        }
      }
      Console.Write("\n");
    }
    Console.ReadKey();
  }
```

Фрагмент кода ***Ch04Ex06\Program.cs***

3. Запустите приложение. На рис. 4.8 показан результат, который должен получиться.

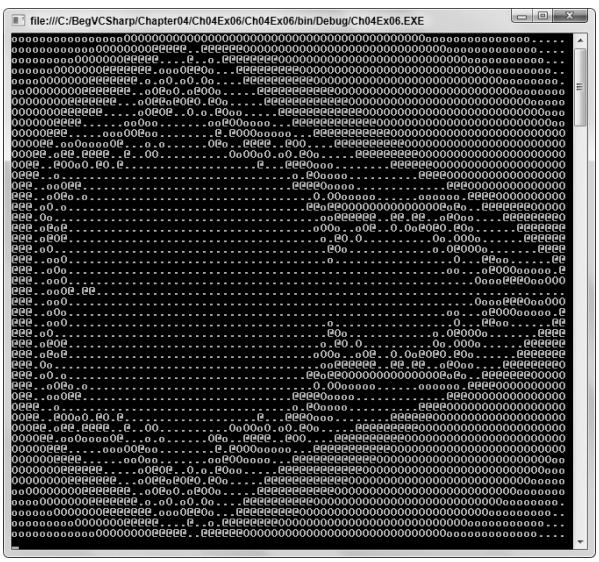


Рис. 4.8. *Результат выполнения приложения* `Ch04Ex06`

Описание работы

Теоретические сведения, необходимые для вычисления множества Мандельброта, выходят за рамки данной главы, но это не помешает понять, что делают циклы в приведенном коде. Те, кому не интересны математические детали, могут спокойно пропустить два следующих абзаца, потому что главное все-таки — понимание кода.

Каждая позиция в изображении Мандельброта соответствует комплексному числу вида `N = x + y*i`, где `x` — вещественная часть `N`, `y` — мнимая, а `i` — квадратный корень из `-1`. Координаты `x` и `y` позиции на изображении соответствуют частям `x` и `y` в комплексном числе.

Для каждой позиции в изображении выполняется проверка аргумента `N`, который равен квадратному корню из `x*x + y*y`. Если его значение больше или равно 2, то позиция, соответствующая данному числу, имеет значение 0. Если же значение аргумента меньше 2, то оно заменяется значением `N*N-N` (что дает `N=(x*x-y*y-x)+(2*x*y-y)*i`) и снова подвергается проверке. Если это значение оказывается больше или равно 2, то позиция, соответствующая этому числу, имеет значение 1. И этот процесс продолжается до тех пор, пока либо позиции в изображении будет присвоено значение, либо будет выполнено определенное количество итераций.

На основании значений, присваиваемых каждой точке в изображении, в графической среде на экране отображался бы пиксель определенного цвета. Однако в данном примере используется текстовый режим, и вместо пикселей на экран выводятся символы.

Теперь вернемся к коду и содержащимся в нем циклам. Начинается код с объявления необходимых для вычислений переменных:

```
double realCoord, imagCoord;
double realTemp, imagTemp, realTemp2, arg;
int iterations;
```

Здесь переменные `realCoord` и `imagCoord` представляют собой вещественную и мнимую часть числа `N`, а остальные переменные типа `double` предназначены для хранения временной информации во время вычислений. Переменная `iterations` будет содержать количество итераций, выполненных перед тем, как значение аргумента `N` (`arg`) станет равно 2 или больше.

Далее выполняются два вложенных цикла `for` для циклического перебора всех координат изображения (с использованием для изменения счетчиков не `++` и `--`, а более сложного синтаксиса, который тоже является весьма распространенным приемом):

```
for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
{
  for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
  {
```

Здесь используются значения, необходимые для отображения основной части множества Мандельброта. Попробуйте варьировать их, чтобы изменить масштаб изображения.

Код внутри этих двух циклов относится только к одной точке в множестве Мандельброта и работает со значением `N`. Здесь вычисляется количество итераций, которое необходимо для отображения текущей точки.

Сначала инициализируются некоторые переменные:

```
iterations = 0;
realTemp = realCoord;
imagTemp = imagCoord;
arg = (realCoord * realCoord) + (imagCoord * imagCoord);
```

Далее следует цикл `while`, в котором выполняются итерации. Цикл `do` не годится, потому что значение `N` сразу может иметь аргумент, больший 2, тогда `iterations = 0`, и никакие вычисления не потребуются.

Обратите внимание, что здесь код не выполняет полное вычисление аргумента, а просто вычисляет значение `x*x-y*y` и проверяет, меньше ли оно 4. Это упрощает вычисления, поскольку известно, что квадратный корень из 4 равен 2, и самостоятельно вычислять другие корни квадратные не требуется:

```
while ((arg < 4) && (iterations < 40))
{
   realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
   imagTemp = (2 * realTemp * imagTemp) - imagCoord;
   realTemp = realTemp2;
   arg = (realTemp * realTemp) + (imagTemp * imagTemp);
   iterations += 1;
}
```

Максимально возможное количество итераций данного цикла равно 40.

После сохранения значения для текущей точки в переменной `iterations` вступает в действие оператор `switch`, который выбирает символ, выводимый на экран. Здесь используются только 4 разных символа вместо 40 возможных, для выбора которых применяется операция `%`: значения 0, 4, 8 и т.д. дают один символ, значения 1, 5, 9 и т.д. — другой и т.п.:

```
   switch (iterations % 4)
   {
      case 0:
         Console.Write(".");
         break;
      case 1:
         Console.Write("o");
         break;
      case 2:
         Console.Write("O");
         break;
      case 3:
      Console.Write("@");
         break;
   }
```

Здесь вместо привычной команды `Console.WriteLine()` применяется команда `Console.Write()` — ведь в данной ситуации не нужно, чтобы каждый выводимый символ отображался в новой строке. Хотя в конце одного из наиболее глубоко вложенных циклов `for` все-таки нужно завершить строку, поэтому в нем просто выводится символ конца строки с помощью уже известной управляющей последовательности:

```
    }
    Console.Write("\n");
}
```

Это обеспечивает отделение строк друг от друга и их выравнивание. Полученный результат, который данное приложение выдает в конечном итоге, конечно, не потрясает, но все-таки впечатляет и уж, конечно, демонстрирует пользу ветвления кода и организации циклов.

Прерывание циклов

Иногда бывает нужна возможность более точного управления выполнением содержащегося в циклах кода. В языке C# для этого предусмотрено четыре команды, три из которых уже применялись в других ситуациях:

- `break` — приводит к немедленному завершению цикла;
- `continue` — приводит к немедленному завершению текущей итерации цикла (выполнение продолжается со следующей итерации);

- `goto` — позволяет выйти из цикла к выполнению кода с указанной меткой (не рекомендуется, если код должен быть легким для чтения и понимания);
- `return` — приводит к выходу из цикла и содержащей его функции (см. главу 6).

Команда `break` просто завершает цикл, после чего выполняется строка кода сразу после этого цикла, например:

```
int i = 1;
while (i <= 10)
{
  if (i == 6)
    break;
  Console.WriteLine("{0}", i++);
}
```

Этот код выведет на консоль числа от 1 до 5, т.к. по достижении переменной `i` значения `6` команда `break` выполняет выход из цикла.

Команда `continue` прерывает лишь текущую итерацию, а не весь цикл:

```
int i;
for (i = 1; i <= 10; i++)
{
  if ((i % 2) == 0)
    continue;
  Console.WriteLine(i);
}
```

В этом примере всякий раз, когда остаток от деления значения `i` на `2` равен нулю, оператор `continue` прерывает выполнение текущей итерации, и на экран выводятся только числа `1`, `3`, `5`, `7` и `9`.

Третьим способом прерывания цикла является использование оператора `goto`:

```
int i = 1;
while (i <= 10)
{
  if (i == 6)
    goto exitPoint;
  Console.WriteLine("{0}", i++);
}
Console.WriteLine("This code will never be reached.");
                // Этот код недостижим
exitPoint:
Console.WriteLine("This code is run when the loop is exited using goto.");
                // Этот код выполняется после выхода из цикла оператором goto
```

Учтите, что выход из цикла с помощью оператора `goto` вполне допустим (хотя и не очень аккуратен), но недопустимо его применение для перехода внутрь цикла извне.

Бесконечные циклы

Существует возможность — ошибочно или преднамеренно — определять так называемые *бесконечные циклы*, т.е. циклы, которые никогда не завершаются. В качестве очень простого примера рассмотрим следующий код:

```
while (true)
{
  // Код тела цикла
}
```

Такой цикл может быть полезен, а выйти из него можно либо с помощью оператора `break`, либо вручную через диспетчер задач Windows. Однако когда подобные циклы получаются случайно, это обычно означает неприятности. Рассмотрим следующий цикл, который похож на цикл `for` из предыдущего раздела:

```
int i = 1;
while (i <= 10)
{
   if ((i % 2) == 0)
      continue;
   Console.WriteLine("{0}", i++);

}
```

Здесь значение переменной i должно увеличиваться только в самой последней строке цикла, которая находится после оператора continue. При срабатывании continue (когда переменная i равна 2) на второй итерации цикла будет использоваться то же самое значение переменной i, что приведет к продолжению цикла, проверке i, снова продолжению цикла и т.д. В конечном итоге это приведет к зависанию приложения. Правда, работу зависшего приложения можно прервать, не перезагружая систему.

Резюме

В данной главе вы пополнили свой инструментарий программиста различными структурами, которые можно использовать в коде. Правильное применение этих структур играет существенную роль при создании более сложных приложений, поэтому они еще не раз будут встречаться в книге.

Сначала в главе рассказывалось о булевской и (немного) поразрядной логике. Теперь, ознакомившись с остальными разделами главы, вы понимаете, насколько важна данная тема при реализации в программах циклов и разветвлений. Уверенное владение описанными в этом разделе операциями и приемами крайне необходимо.

Разветвления позволяют выполнять части кода в зависимости от выполнения тех или иных условий. Вместе с циклами их можно применять для создания в коде C# разнообразных сложных структур. При наличии циклов внутри других циклов, находящихся в структурах if, которые вложены еще в какие-то циклы, вы быстро поймете, почему так полезны отступы в коде. Если прижать весь код влево, то он станет трудным для чтения и еще более трудным для отладки. Очень важно уже на этом этапе хорошенько разобраться с использованием отступов, потом это обязательно окупится. Среда VS добавляет многие отступы автоматически, но все равно лучше освоить это дело и самостоятельно вставлять отступы при вводе кода.

Следующая глава посвящена более детальному изучению переменных.

Упражнения

1. Пусть имеются два целых числа, хранящихся в переменных var1 и var2. Какая логическая проверка позволяет выяснить, является ли одно из них (но не оба вместе) больше 10?
2. Напишите приложение, включающее логику из первого упражнения, которое запрашивает у пользователя два числа и выводит их на экран. Но если оба числа больше 10, оно должно отклонить их и предложить пользователю ввести два других числа.
3. Что неверно в следующем коде?

```
int i;
for (i = 1; i <= 10; i++)
{
   if ((i % 2) = 0)
      continue;
   Console.WriteLine(i);
}
```

4. Измените приложение, отображающее множество Мандельброта, так, чтобы оно запрашивало у пользователя границы изображения и выводило только указанную часть рисунка. Приведенный в тексте главы код выводит столько символов, сколько сможет уместиться в одной строке консольного приложения; подумайте, как сделать так, чтобы каждый выбираемый рисунок занимал такое же место, максимально увеличив видимую область.

Решения упражнений приведены в приложении A.

Что вы узнали в этой главе

Тема	Основные концепции
Булевская логика	Для вычисления условий в булевской логике используются булевские значения — `true` и `false`. Для сравнения логических значений применяются булевские операции, которые возвращают булевские результаты. Некоторые булевские операции используются также для выполнения поразрядных действий над двоичными представлениями значений, а, кроме того, имеются и специальные поразрядные операции.
Ветвление	Булевскую логику можно использовать для управления потоком выполнения программы. Результат выражения, которое вычисляет логическое значение, можно применять для определения, нужно ли выполнять некоторых блок кода. Простые разветвления выполняются с помощью операторов `if` или тернарных операций `?:`, а одновременные проверки нескольких условий — с помощью операторов `switch`.
Циклы	Циклы позволяют выполнять блоки кода несколько раз, в зависимости от указанных условий. С помощью циклов `do` и `while` можно выполнять код до тех пор, пока некоторое логическое выражение не возвратит `true`, а циклы `for` позволяют добавить в циклически повторяемый код счетчик. Можно прервать как текущую итерацию цикла (с помощью оператора `continue`), так и полностью весь цикл (с помощью оператора `break`). Некоторые циклы могут завершиться только по прерыванию, они называются бесконечными циклами.

5

Дополнительные сведения о переменных

В ЭТОЙ ГЛАВЕ...

- Выполнение неявных и явных преобразований типов
- Создание и использование типов `enum`
- Создание и использование типов `struct`
- Создание и использование массивов
- Работа со строковыми значениями

Теперь, когда вы уже немного освоились с языком C#, можно вернуться к более сложным вопросам, касающимся переменных.

Сначала мы познакомимся с *преобразованием типов*, т.е. с переводом значений из одного типа в другой. Данный вопрос уже встречался ранее, но здесь он будет рассмотрен более формально. Это позволит лучше понять, что происходит при (намеренном или случайном) смешивании типов в выражениях, а также более грамотно управлять процессом обработки данных, что поможет упростить код и избежать неприятных сюрпризов.

Далее будут рассмотрены еще несколько возможных типов переменных.

- **Перечисления**. Типы переменных, которые могут содержать определяемый пользователем дискретный набор возможных значений — они позволяют сделать код более понятным.
- **Структуры**. Составные типы переменных, состоящие из определенного пользователем набора переменных различных типов.
- **Массивы**. Типы, которые хранят несколько переменных одного типа и предоставляют индексный доступ к отдельным значениям.

Эти типы немного сложнее уже знакомых нам простых типов, которые демонстрировались ранее, но они могут значительно упростить жизнь. И в завершение главы будет рассмотрена еще одна полезная тема — простые приемы обработки строк.

Преобразование типов

Ранее в книге уже показывалось, что все данные, независимо от типа, представляют собой последовательность битов, т.е. последовательность нулей и единиц. Смысл переменной определяется способом интерпретации этих данных. Наиболее простым примером является тип `char`. Этот тип представляет символ из набора символов Unicode с использованием числа. На самом деле он хранится в памяти точно так же, как тип `ushort`, поскольку они оба могут хранить числа от 0 до 65 535.

Однако в общем случае для разных типов переменных применяются разные схемы для представления данных. Это означает, что даже если бы можно было помещать последовательность битов из одной переменной в переменную другого типа (если обе занимают одинаковый объем памяти или если принимающий тип может вместить все биты из исходного типа), результаты оказались бы не обязательно такими, какие можно было ожидать.

Вместо такого взаимно однозначного копирования битов из одной переменной в другую необходимо применять *преобразования типов*. Они бывают двух видов.

- **Неявное преобразование**. Применяется, когда преобразование из типа A в тип B возможно при любых обстоятельствах, а правила выполнения преобразования достаточно просты для того, чтобы доверить их компилятору.
- **Явное преобразование**. Применяется, когда преобразование из типа A в тип B возможно только при определенных обстоятельствах или когда правила преобразования довольно сложны и требуют дополнительной обработки.

Неявные преобразования

Неявное преобразование не требует от разработчика ни дополнительных усилий, ни дополнительного кода. Рассмотрим следующий код:

```
var1 = var2;
```

Эта операция присваивания может выполнять неявное преобразование, если тип `var2` может быть неявно преобразован в тип `var1` — или не выполнять, если типы обеих переменных одинаковы. Например, значения типов `ushort` и `char` по существу взаимозаменяе-

мы, поскольку оба этих типа позволяют хранить числа от 0 до 65 535. Поэтому между ними может осуществляться неявное преобразование, как показано в следующем коде:

```
ushort destinationVar;
char sourceVar = 'a';
destinationVar = sourceVar;
Console.WriteLine("значение sourceVar: {0}", sourceVar);
Console.WriteLine("значение destinationVar: {0}", destinationVar);
```

Здесь значение, хранящееся в `sourceVar`, помещается в `destinationVar`, а после вывода переменных с помощью двух команд `Console.WriteLine()` на экране появляется следующий результат:

```
значение sourceVar: a
значение destinationVar: 97
```

Хотя в этих двух переменных хранится одинаковая информация, интерпретируются они разными способами, зависящими от их типа.

Неявное преобразование поддерживается между многими простыми типами; правда, для типов `bool` и `string` это невозможно, но зато для числовых типов имеется несколько вариантов. Для справки в табл. 5.1 перечислены все числовые преобразования, которые неявно может выполнять компилятор (поскольку значения типа `char` сохраняются как числа, тип `char` тоже считается числовым).

Таблица 5.1. Неявные числовые преобразования

Тип	Может быть безопасно преобразован в тип
`byte`	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`
`sbyte`	`short`, `int`, `long`, `float`, `double`, `decimal`
`short`	`int`, `long`, `float`, `double`, `decimal`
`ushort`	`int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`
`int`	`long`, `float`, `double`, `decimal`
`uint`	`long`, `ulong`, `float`, `double`, `decimal`
`long`	`float`, `double`, `decimal`
`ulong`	`float`, `double`, `decimal`
`float`	`double`
`char`	`ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`

Не волнуйтесь, учить эту таблицу наизусть не нужно, поскольку сообразить, какие преобразования компилятор может выполнять неявно, на самом деле довольно просто. В главе 3 была приведена таблица с диапазонами возможных значений для каждого простого числового типа. Правило неявного преобразования для этих типов гласит, что любой тип A, диапазон возможных значений которого полностью вписывается в диапазон возможных значений типа B, может быть преобразован в этот тип неявным образом.

Объясняется это очень просто. Попытка уместить в переменную значение, выходящее за диапазон значений, которые переменная этого типа может принимать, приведет к проблеме. Например, переменная типа `short` способна хранить числа до 32 767, а максимальным значением, которое может храниться в переменной типа `byte`, является 255 — поэтому при попытке преобразовать значение `short` в `byte` могут возникнуть проблемы. Ведь если значением в переменной `short` окажется число от 256 до 32 767, то оно просто не поместится в переменную `byte`.

Но если известно, что значение в переменной типа `short` не больше 255, то тогда должен быть способ его преобразования в `char`? Простым ответом, будет, конечно же, да, а более сложным — можно, но с использованием *явного* преобразования. Явное преобразование означает, что вы берете ответственность за происходящее на себя.

Явные преобразования

Уже по названию понятно, что явное преобразование выполняется тогда, когда разработчик явным образом просит компилятор преобразовать значение из одного типа данных в другой. Для этого требуется дополнительный код, формат которого может выглядеть поразному, в зависимости от конкретного метода преобразования. Прежде чем рассматривать какой-либо код с явным преобразованием, следует посмотреть, что будет, если его не добавлять.

Например, ниже приведен измененный код из предыдущего раздела, в котором предпринимается попытка преобразовать значение `short` в `byte`:

```
byte destinationVar;
short sourceVar = 7;
destinationVar = sourceVar;
Console.WriteLine("значение sourceVar: {0}", sourceVar);
Console.WriteLine("значение destinationVar: {0}", destinationVar);
```

При попытке скомпилировать этот код будет выдана следующая ошибка:

```
Cannot implicitly convert type 'short' to 'byte'. An explicit conversion exists
(are you missing a cast?)
Неявное преобразование типа 'short' в 'byte' невозможно. Существует явное
преобразование (возможно, пропущено приведение)
```

К счастью, компилятор C# способен выявлять недостающие операции явного преобразования типов.

Чтобы этот код компилировался, в него необходимо добавить код, выполняющий явное преобразование. Наиболее простой способ — *приведение* (cast) переменной типа `short` к типу `byte` (как и было предложено в сообщении об ошибке). Операция приведения, по сути, заключается в принудительном преобразовании данных из одного типа в другой с помощью следующего простого синтаксиса:

```
(целевойТип) исходнаяПеременная
```

Это выражение преобразовывает значение, содержащееся в *исходнаяПеременная*, в значение типа *целевойТип*.

Приведение типов возможно не во всех ситуациях. Типы, мало похожие или вовсе непохожие друг на друга, скорее всего, не будут поддерживать преобразование приведением типов.

С помощью этого синтаксиса можно изменить пример так, чтобы в нем выполнялось принудительное преобразование из `short` в `byte`:

```
byte destinationVar;
short sourceVar = 7;
destinationVar = (byte)sourceVar;
Console.WriteLine("значение sourceVar: {0}", sourceVar);
Console.WriteLine("значение destinationVar: {0}", destinationVar);
```

После этого выходные данные будут выглядеть так:

```
значение sourceVar: 7
значение destinationVar: 7
```

А что произойдет, если выполнить преобразование значения в переменную, в которой оно не сможет уместиться? Изменим предыдущий код, как показано ниже:

```
byte destinationVar;
short sourceVar = 281;
destinationVar = (byte)sourceVar;
Console.WriteLine("значение sourceVar: {0}", sourceVar);
Console.WriteLine("значение destinationVar: {0}", destinationVar);
```

Это приведет к получению следующего результата:

```
значение sourceVar: 281
значение destinationVar: 25
```

Что произошло? Чтобы понять это, рассмотрим двоичные представления двух полученных чисел, а также числа 255 — максимального значения, которое может находиться в переменной типа `byte`:

```
281 = 100011001
 25 = 000011001
255 = 011111111
```

Видно, что потерян самый левый бит исходных данных. Сразу же возникает вопрос: как узнать, когда такое может произойти? Очевидно, что ситуации, когда нужно явно приводить один тип к другому, обязательно будут встречаться, и потому необходимо как-то определять, привело ли приведение к потере данных. В противном случае возможны очень серьезные ошибки — а если это приложение вроде банковского учета или вычисления траектории полета ракеты на Луну?

Один из способов такого определения состоит в сравнении значения исходной переменной с известными предельными значениями целевой переменной. Еще один способ — указание системе обратить особое внимание на данную операцию преобразования во время выполнения. Попытка уместить слишком большое значение в переменной приводит к *переполнению*, а возникновение переполнения можно отследить.

Для установки так называемого *контекста проверки на переполнение* в выражении имеются два ключевых слова: `checked` и `unchecked`. Применяются они следующим образом:

```
checked(<выражение>)
unchecked(<выражение>)
```

В нашем примере указать необходимость проверки на переполнение можно так:

```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("значение sourceVar: {0}", sourceVar);
Console.WriteLine("значение destinationVar: {0}", destinationVar);
```

Попытка выполнения этого кода приведет к аварийному завершению и выводу сообщения об ошибке — см. рис. 5.1, который был получен после компиляции проекта `OverflowCheck`.

Но если заменить в этом коде `checked` на `unchecked`, то он выдаст точно такой же результат, как и раньше, без сообщений об ошибках. Такое поведение идентично уже продемонстрированному ранее поведению по умолчанию.

Кроме того, можно настроить все приложение, чтобы оно вело себя так, будто перед каждым подобным выражением находится слово `checked`, если только в этом выражении явно не записано слово `unchecked` (т.е. изменить стандартную настройку для проверок на переполнения). Для этого нужно изменить свойства проекта: щелкните правой кнопкой мыши на проекте в окне Solution Explorer (Проводник решений), выберите в контекстном меню пункт Properties (Свойства) и выберите элемент Build (Сборка) в левой части окна. Это приведет к отображению параметров сборки, как показано на рис. 5.2.

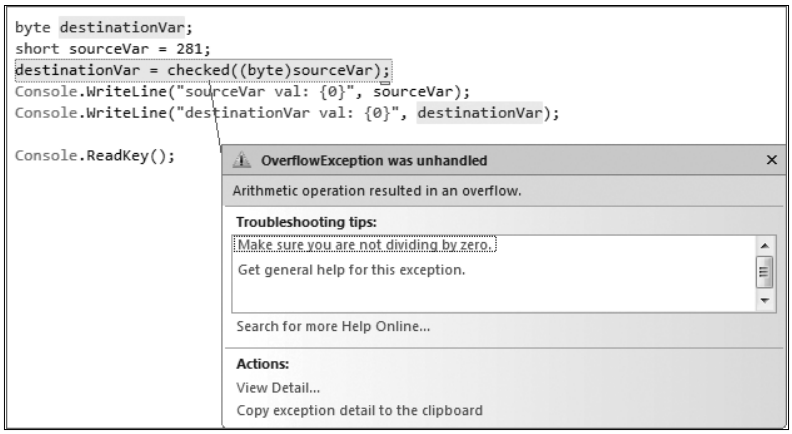


Рис. 5.1. *Сообщение об обнаруженном переполнении*

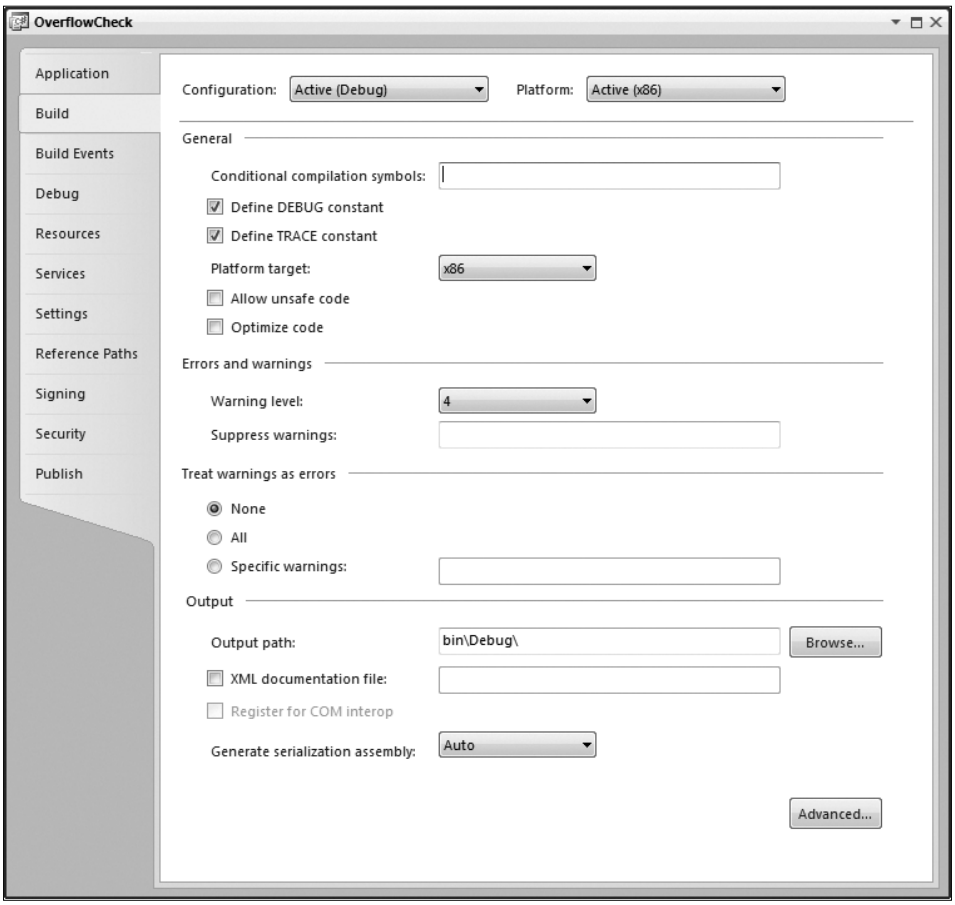


Рис. 5.2. *Параметры сборки*

Свойство, которое необходимо изменить, относится к дополнительным параметрам, поэтому щелкните на кнопке Advanced (Дополнительно) и в открывшемся диалоговом окне установите флажок Check for arithmetic overflow/underflow (Выполнять проверку на арифметическое переполнение/потерю значимости), как показано на рис. 5.3. По умолчанию этот флажок сброшен, а его установка обеспечивает описанное поведение `checked`.

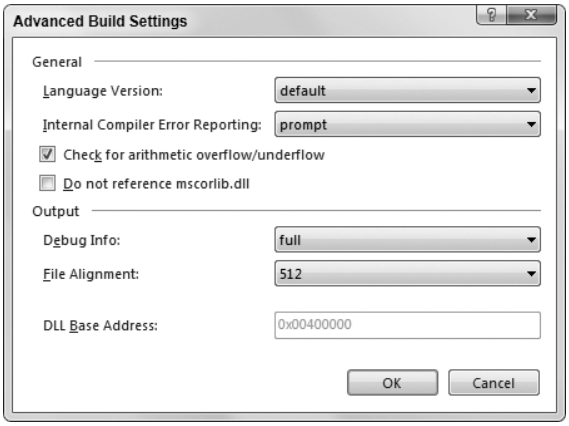


Рис. 5.3. *Установка флажка* Check for arithmetic overflow/underflow option

Явные преобразования с помощью команд `Convert`

Операции явного преобразования, которые использовались ранее в практических занятиях, немного отличаются от рассмотренных в настоящей главе. Там строковые значения преобразовывались в числа с помощью команд вроде `Convert.ToDouble()` — понятно, что такие команды не будут работать для всех возможных строк.

Например, попытка преобразовать строку `Number` в значение типа `double` с помощью команды `Convert.ToDouble()` приведет при выполнении кода к появлению диалогового окна, показанного на рис. 5.4.

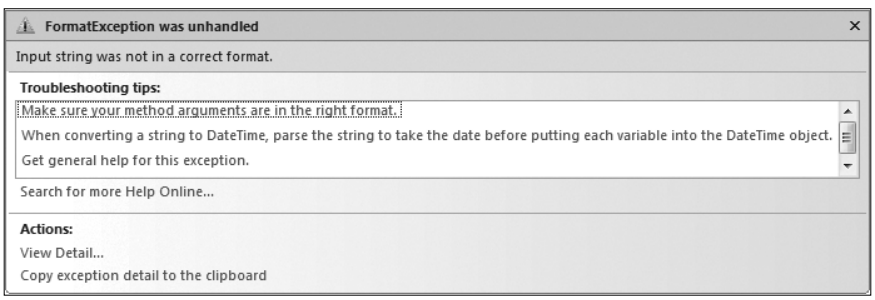


Рис. 5.4. *Сообщение о попытке преобразования недопустимой строки в число*

Как видите, операция не была выполнена. Для работы таких преобразований передаваемая им строка *обязательно* должна содержать допустимое представление числа — причем такого числа, которое не приведет к переполнению. Допустимое представление числа должно содержать (необязательный) знак (плюс или минус), ноль или более цифр, необязательную точку, за которой следует еще одна или более цифр, символ `e` или `E`, за которым следует необязательный знак и одна или более цифр — и *больше ничего, кроме символов пробела*

(перед и после этой последовательности). В случае использования всех необязательных элементов допустимые строки могут иметь вид наподобие `-1.2451e-24`.

Существует много таких явных операций преобразования, все они приведены в табл. 5.2.

Таблица 5.2. Команды для явных преобразований

Команда	Результат
`Convert.ToBoolean(val)`	`val`, преобразованное в `bool`
`Convert.ToByte(val)`	`val`, преобразованное в `byte`
`Convert.ToChar(val)`	`val`, преобразованное в `char`
`Convert.ToDecimal(val)`	`val`, преобразованное в `decimal`
`Convert.ToDouble(val)`	`val`, преобразованное в `double`
`Convert.ToInt16(val)`	`val`, преобразованное в `short`
`Convert.ToInt32(val)`	`val`, преобразованное в `int`
`Convert.ToInt64(val)`	`val`, преобразованное в `long`
`Convert.ToSByte(val)`	`val`, преобразованное в `sbyte`
`Convert.ToSingle(val)`	`val`, преобразованное в `float`
`Convert.ToString(val)`	`val`, преобразованное в `string`
`Convert.ToUInt16(val)`	`val`, преобразованное в `ushort`
`Convert.ToUInt32(val)`	`val`, преобразованное в `uint`
`Convert.ToUInt64(val)`	`val`, преобразованное в `ulong`

Здесь на месте `val` может быть указана переменная практически любого типа (если данная команда не работает с указанным типом, компилятор сообщит об этом).

К сожалению, имена команд преобразования, приведенных в табл. 5.2, несколько отличаются от используемых в C# имен типов: например, для выполнения преобразования в тип `int` нужна команда `Convert.ToInt32()`. Объясняется это тем, что все эти команды взяты из пространства имен `System` в .NET Framework и не являются родными для C#. Зато их можно использовать не только в C#, но и в других совместимых с .NET языках.

Учтите, что в этих командах преобразования *всегда* выполняется проверка на переполнение, и ни ключевые слова `checked` и `unchecked`, ни свойства проекта на них никак не влияют.

В следующем практическом занятии демонстрируется применение большинства из рассмотренных в этом разделе видов преобразования. В нем сначала объявляется и инициализируется несколько переменных различных типов, а затем выполняются операции преобразования между их типами как неявным, так и явным образом.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с преобразованием типов

1. Создайте новое консольное приложение с именем `Ch05Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter05`.
2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
  short shortResult, shortVal = 4;
```

```
        int integerVal = 67;
        long longResult;
        float floatVal = 10.5F;
        double doubleResult, doubleVal = 99.999;
        string stringResult, stringVal = "17";
        bool boolVal = true;

        Console.WriteLine("Variable Conversion Examples\n");
                        // Примеры преобразования переменных

        doubleResult = floatVal * shortVal;
        Console.WriteLine("Implicit, -> double: {0} * {1} - > {2}",
          floatVal, shortVal, doubleResult);     // Неявное преобразование

        shortResult = (short)floatVal;
        Console.WriteLine("Explicit, -> short:  {0} - > {1}",
          floatVal, shortResult);                // Явное преобразование

        stringResult = Convert.ToString(boolVal) + Convert.ToString(doubleVal);
        Console.WriteLine("Explicit, -> string: \"{0}\" + \"{1}\" - > {2}",
          boolVal, doubleVal, stringResult);     // Явное преобразование

        longResult = integerVal + Convert.ToInt64(stringVal);
        Console.WriteLine("Mixed,    -> long:   {0} + {1} - > {2}",
          integerVal, stringVal, longResult);    // Смешанное преобразование
        Console.ReadKey();
    }
```

Фрагмент кода ***Ch05Ex01\Program.cs***

3. Запустите приложение. На рис. 5.5 показан результат, который должен получиться.

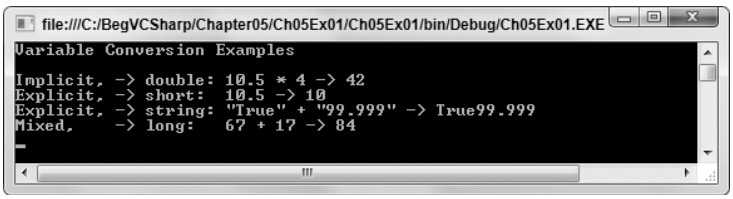


Рис. 5.5. *Приложение* `Ch05Ex01` *в действии*

Описание работы

В этом примере присутствуют все уже рассмотренные виды преобразований — как в простых операторах присваивания, подобных приведенным ранее в коротких примерах, так и в выражениях. Необходимо разобраться в обоих случаях, поскольку преобразование типов может выполняться не только в операциях присваивания, но и в *любой* не унарной операции. Например:

```
shortVal * floatVal
```

Здесь значение `short` умножается на значение `float`. В таких ситуациях, где не заданы явные преобразования, по возможности будет выполняться неявное преобразование. В данном примере единственным разумным неявным преобразованием является преобразование типа `short` в тип `float` (для преобразования `float` в `short` требуется явная операция), поэтому именно оно и будет использоваться.

Но при необходимости это поведение можно переопределить:

```
shortVal * (short)floatVal
```

Учтите, что перемножение двух значений типа `short` *не возвращает значение типа* `short`*. Из-за того, что результат умножения вполне может превышать 32 767 (максимальное значение, которое может храниться в переменных типа* `short`*), возвращаемый результат имеет тип* `int`*.*

Явные преобразования, выполняемые с помощью такого синтаксиса приведения, имеют тот же приоритет, что и другие унарные операции (вроде префиксной операции ++) — т.е. высший.

При наличии операторов, содержащих смешанные типы, преобразование происходит при выполнении каждой операции, в соответствии с приоритетами. Это означает, что могут иметь место и промежуточные преобразования. Например:

```
doubleResult = floatVal + (shortVal * floatVal);
```

Здесь первой будет выполнена операция `*`, но перед этим, как уже было сказано, `shortVal` будет преобразовано во `float`. Затем будет выполнена операция `+`, которая не требует преобразования типов, поскольку имеет дело с двумя значениями `float` (`floatVal` и результат вычисления `shortVal * floatVal`). И последней будет выполнена операция `=`, перед которой результат предыдущей операции, имеющий тип `float`, будет преобразован в тип `double`.

Такой процесс преобразования может показаться довольно сложным, но разбиение выражений на отдельные части с учетом приоритетов операций поможет разобраться, что к чему.

Составные типы переменных

Кроме всех простых типов переменных, в C# имеются и три немного более сложных (но очень полезных) вида переменных: перечисления, структуры и массивы.

Перечисления

У каждого из рассмотренных до сих пор типов (за исключением `string`) имеется четко определенный набор допустимых значений. Правда, у типов вроде `double` он настолько велик, что его можно считать бесконечным, но *все равно* это фиксированный набор. Самым простым примером является тип `bool`, который может принимать только одно из двух значений — `true` или `false`.

Но существует и много других ситуаций, в которых может понадобиться переменная, способная принимать одно значение из фиксированного набора возможных значений. Например, может потребоваться переменная типа `orientation`, способная хранить одно из значений `north`, `south`, `east` или `west` (север, юг, восток, запад).

В таких ситуациях применяются *перечисления*. Перечисления предназначены как раз для того, что нужно в случае `orientation`: они позволяют определять тип, способный принимать одно значение из заданного набора. Тогда нужно лишь создать переменную типа `orientation`, которая может принимать одно из четырех возможных значений.

Однако учтите, что здесь необходим еще один дополнительный шаг: нужно не просто объявлять переменную конкретного типа, а сначала объявить и описать пользовательский тип и только затем объявить переменную этого типа.

Определение перечислений

Перечисления определяются с помощью ключевого слова `enum` следующим образом:

```
enum имя_типа
{
   <значение1>,
```

```
    <значение2>,
    <значение3>,
    ...
    <значениеN>
}
```

После этого можно объявлять переменные этого нового типа с помощью обычного синтаксиса:

```
<имяТипа> <имяПеременной>;
```

Присваивать значения таким переменным можно следующим образом:

```
<имяПеременной> = <имяТипа>.<значение>;
```

У перечислений имеется *базовый тип*, используемый для хранения. Каждое из значений, которое может приниматься типом перечисления, хранится в виде значения данного базового типа — по умолчанию это `int`. Указать другой базовый тип можно, добавив желаемый тип в объявление перечисления:

```
enum <имяТипа>: <базовыйТип>
{
    <значение1>,
    <значение2>,
    <значение3>,
    ...
    <значениеN>
}
```

Для перечислений могут использоваться базовые типы `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong`.

По умолчанию каждому значению перечисления автоматически присваивается значение соответствующего базового типа в соответствии с порядком их определения, начиная с нуля. Это означает, что *<значение1>* ставится в соответствие 0, *<значение2>* — 1, *<значение3>* — 2 и т.д. Это присваивание можно переопределить с помощью операции = и указания фактических желаемых значений для каждого значения перечисления:

```
enum <имяТипа>: <базовыйТип>
{
    <значение1> = <фактическоеЗначение1>,
    <значение2> = <фактическоеЗначение2>,
    <значение3> = <фактическоеЗначение3>,
    ...
    <значениеN> = <фактическоеЗначениеN>
}
```

Помимо этого, можно указывать идентичные значения для нескольких значений перечисления с использованием одного значения в качестве базового для другого значения:

```
enum <имяТипа>: <базовыйТип>
{
    <значение1> = <фактическоеЗначение1>,
    <значение2> = <значение1>,
    <значение3>,
    ...
    <значениеN> = <фактическоеЗначениеN>,
}
```

Всем значениям, которым ничего явно присвоено, автоматически назначаются базовые значения, большие на 1, 2 и т.д., чем у последнего явно заданного. Например, в приведенном выше коде *<значение3>* будет равно *<значение1>* + 1.

Указание одинаковых значений наподобие *<значение2> = <значение1>* может привести к проблемам. Например, в следующем коде *<значение4>* получит точно такое же значение, что и *<значение2>*:

```
enum <имяТипа>: <базовыйТип>
{
   <значение1> = <фактическоеЗначение1>,
   <значение2>,
   <значение3> = <значение1>,
   <значение4>,
   ...
   <значениеN> = <фактическоеЗначениеN>,
}
```

Конечно, если такое поведение как раз и нужно, то тут все в порядке. Учтите, что циклическое присваивание значений приведет к ошибке:

```
enum <имяТипа>: <базовыйТип>
{
   <значение1> = <значение2>,
   <значение2> = <значение1>
}
```

Все это демонстрируется в следующем практическом занятии. В нем определяется, а затем используется перечисление с именем `orientation`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование перечисления

1. Создайте новое консольное приложение с именем `Ch05Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter05`.
2. Добавьте в файл `Program.cs` следующий код:

```
namespace Ch05Ex02
{
   enum orientation : byte
   {
      north = 1,
      south = 2,
      east = 3,
      west = 4
   }
   class Program
   {
      static void Main(string[] args)
      {
         orientation myDirection = orientation.north;
         Console.WriteLine("myDirection = {0}", myDirection);
         Console.ReadKey();
      }
   }
}
```

*Фрагмент кода **Ch05Ex02\Program.cs***

3. Запустите приложение. На рис. 5.6 показан вывод, который должен получиться.
4. Выйдите из приложения и измените его код следующим образом:

```
byte directionByte;
string directionString;
orientation myDirection = orientation.north;
```

```
Console.WriteLine("myDirection = {0}", myDirection);
directionByte = (byte)myDirection;
directionString = Convert.ToString(myDirection);
Console.WriteLine("byte equivalent = {0}", directionByte);
                // байтовый эквивалент
Console.WriteLine("string equivalent = {0}", directionString);
                // строковый эквивалент
Console.ReadKey();
```

5. Запустите приложение снова. Теперь вывод должен получиться таким, как показано на рис. 5.7.

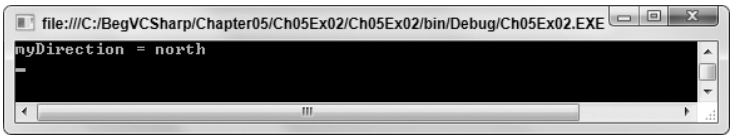


Рис. 5.6. *Приложение* `Ch05Ex02` *в действии*

Рис. 5.7. *Вывод модифицированного приложения* `Ch05Ex02`

Описание работы

Приведенный код демонстрирует определение и применение типа перечисления с именем `orientation`. Первое, на что нужно обратить внимание — код определения типа размещен в отдельном пространстве имен `Ch05Ex02`, а не в том же месте, что и остальная часть кода. Объясняется это тем, что определения не выполняются, т.е. во время выполнения поток управления не проходит по коду определения, а только по строкам кода остальной части приложения. Приложение начинает выполняться в обычном месте и имеет доступ к новому типу — ведь оно относится к тому же пространству имен.

Первый вариант примера демонстрирует базовый метод создания переменной нового типа, присваивание ей значения и вывод этого значения на экран. Затем код был изменен для демонстрации преобразования значений перечисления в другие типы. Обратите внимание, что здесь необходимы явные преобразования. Хотя базовым типом `orientation` и является `byte`, для преобразования значения `myDirection` в `byte` все равно нужно использовать приведение `(byte)`:

```
directionByte = (byte)myDirection;
```

Такое же явное приведение необходимо и в обратном направлении, т.е. при необходимости преобразовать `byte` в `orientation`. Например, чтобы преобразовать переменную типа `byte` с именем `myByte` в тип `orientation` и присвоить полученное значение переменной `myDirection`, можно воспользоваться следующим оператором:

```
myDirection = (orientation)myByte;
```

Конечно, здесь нужна осторожность, ведь не все допустимые значения переменной типа `byte` отображаются на значения, указанные в определении переменной `orientation`. Тип `orientation` может хранить и другие значения типа `byte`, и непосредственно при при-

сваивании ошибка не возникнет, но это может привести к нарушению логики при дальнейшей работе приложения.

Для получения строковой версии значения перечисления можно применять команду `Convert.ToString()`:

```
directionString = Convert.ToString(myDirection);
```

Операция приведения `(string)` здесь не работает, поскольку необходимо не просто помещение данных из переменной типа перечисления в переменную типа `string`. Возможен и другой вариант — команда `ToString()` самой переменной. Следующий код даст тот же результат, что и вызов `Convert.ToString()`:

```
directionString = myDirection.ToString();
```

Преобразование строки в значение перечисления тоже возможно, но только требуемый для этого синтаксис выглядит немного сложнее. Для выполнения подобного преобразования рода предусмотрена специальная команда `Enum.Parse()`, которая применяется следующим образом:

```
(<типПеречисления>)Enum.Parse(typeof(<типПеречисления>),
<строкаЗначенияПеречисленния>);
```

Здесь используется еще одна операция — `typeof`, которая получает информацию о типе своего операнда. В случае типа `orientation` она применяется так:

```
string myString = "north";
orientation myDirection = (orientation)Enum.Parse(typeof(orientation),
myString);
```

Разумеется, не все строковые значения отображаются на значения `orientation`. При передаче значения, которое не отображается ни на одно из значений перечисления, возникнет ошибка. Как и все остальное в языке C#, такие значения чувствительны к регистру символов, поэтому ошибка возникнет даже в том случае, если строка совпадает со значением во всем, кроме регистра (например, когда `myString` содержит текст `North`, а не `north`).

Структуры

Структуры (struct) — это действительно структуры данных, состоящие из нескольких фрагментов данных, возможно, даже разного типа. Они позволяют определять собственные типы переменных указанной структуры. Например, предположим, что требуется сохранить информацию о маршруте к месту назначения из начальной точки, а эта информация состоит из отрезков пути, которые определяются направлением и расстоянием (в милях). Для простоты предположим, что направление соответствует сторонам света и, значит, может быть представлено с помощью приведенного в предыдущем разделе перечисления `orientation`, и что расстояние в милях может быть представлено типом `double`.

Тогда для хранения этих сведений можно использовать две отдельных переменных:

```
orientation myDirection;
double      myDistance;
```

Ничего неправильного в применении двух таких переменных нет, но все-таки гораздо проще (особенно при необходимости обработки множества маршрутов) хранить эту информацию в одном месте.

Определение структур

Структуры определяются с помощью ключевого слова `struct`:

```
struct <имяТипа>
{
```

```
    <объявленияЧленов>
}
```

В разделе <*объявленияЧленов*> содержатся объявления переменных (называемых *данными-членами* структуры) в практически том же формате, что и обычно. Объявление каждого члена имеет следующий вид:

```
<уровеньДоступа> <тип> <имя>;
```

Чтобы вызывающий структуру код имел доступ к ее данным-членам, на месте <*уровеньДоступа*> записывается ключевое слово `public`. Например:

```
struct route
{
  public orientation direction;
  public double      distance;
}
```

После определения типа структуры можно определять переменные нового типа:

```
route myRoute;
```

Кроме того, можно обращаться к данным-членам этой составной переменной с помощью символа точки:

```
myRoute.direction = orientation.north;
myRoute.distance = 2.5;
```

Все это демонстрируется в следующем практическом занятии, где используются уже определенные перечисление `orientation` и структура `route`, над которой в коде проводятся различные действия, позволяющие получить представление о работе структуры.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование структуры

1. Создайте новое консольное приложение с именем `Ch05Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter05`.

2. Добавьте в файл `Program.cs` следующий код:

```
namespace Ch05Ex03
{
  enum orientation : byte
  {
    north = 1,
    south = 2,
    east  = 3,
    west  = 4
  }
  struct route
  {
    public orientation direction;
    public double      distance;
  }
  class Program
  {
    static void Main(string[] args)
    {
      route myRoute;
      int myDirection = -1;
      double myDistance;
      Console.WriteLine("1) North\n2) South\n3) East\n4) West");
      do
```

```
        {
          Console.WriteLine("Select a direction:");
                          // Выберите направление:
          myDirection = Convert.ToInt32(Console.ReadLine());
        }
        while ((myDirection < 1) || (myDirection > 4));
        Console.WriteLine("Input a distance:");
                        // Введите расстояние
        myDistance = Convert.ToDouble(Console.ReadLine());
        myRoute.direction = (orientation)myDirection;
        myRoute.distance = myDistance;
        Console.WriteLine("myRoute specifies a direction of {0} and a " +
          "distance of {1}", myRoute.direction, myRoute.distance);
                        // Вывод информации о направлении и расстоянии
        Console.ReadKey();
      }
    }
}
```

Фрагмент кода ***Ch05Ex03\Program.cs***

3. Запустите приложение, а затем укажите направление (число от 1 до 4) и расстояние. На рис. 5.8 показан результат, который должен получиться.

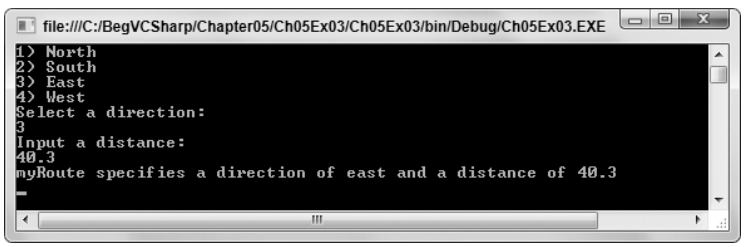


Рис. 5.8. *Приложение* `Ch05Ex03` *в действии*

Описание работы

Структуры, как и перечисления, объявляются за пределами основного тела кода. В данном примере структура `route` объявлена прямо внутри того же пространства имен, что и перечисление `orientation`, которое в ней используется:

```
enum orientation : byte
{
  north = 1,
  south = 2,
  east  = 3,
  west  = 4
}
struct route
{
  public orientation direction;
  public double     distance;
}
```

Как и в предыдущем примере, основное тело кода следует за структурой; в нем выполняется запрос входных данных у пользователя, а затем их вывод. Вводимые данные подвергаются простой проверке: код выбора направления помещается в цикл `do`, который отбрасывает все входные данные, не являющиеся целым числом от 1 до 4 (чтобы они ото-

бражались на члены перечисления и тем самым упрощали выполнение операций присваивания).

Входные данные, которые не будут распознаны как целое число, приводят к ошибке. Почему так происходит и что с этим делать, будет рассказано позже.

Обратите внимание на интересный момент: при обращении к членам `route` они обрабатываются в точности так же, как и обычные переменные того же типа. Операции присваивания выглядят следующим образом:

```
myRoute.direction = (orientation)myDirection;
myRoute.distance = myDistance;
```

Вообще-то входное значение можно сразу записать в `myRoute.distance` без всяких негативных последствий:

```
myRoute.distance = Convert.ToDouble(Console.ReadLine());
```

Просто дополнительный шаг позволяет выполнить дополнительную проверку входных данных, хотя в коде такой проверки нет. Любой доступ к членам структуры выполняется таким же образом. Выражения вида *<переменнаяСтруктуры>.<переменнаяЧлен>* обрабатываются так же, как и обычная переменная типа *<переменнаяЧлен>*.

Массивы

У всех рассмотренных до этого типов была одна общая черта: каждый из них позволял хранить лишь одно значение (или единственный набор значений в случае структур). Однако бывают ситуации, в которых требуется хранить большие объемы данных, а для этого такие типы переменных мало пригодны. Бывает необходимо хранить сразу несколько значений одинакового типа без использования для каждого из них отдельной переменной.

Например, предположим, что требуется выполнить какую-то обработку имен всех ваших друзей. Можно объявить простые строковые переменные:

```
string friendName1 = "Robert Barwell";
string friendName2 = "Mike Parry";
string friendName3 = "Jeremy Beacock";
```

Но такой подход потребует много усилий, особенно из-за того, что для обработки каждой переменной придется писать отдельный код. Организовать перебор этого списка строк в цикле не получится.

Однако существует другой способ — *массивы* (array). Массив — это индексированный список переменных, хранящийся в единой переменной типа массива. Например, для хранения трех показанных выше имен можно создать массив с именем `friendNames`. Для обращения к отдельным членам массива нужно указать их индекс в квадратных скобках:

```
friendNames[<индекс>]
```

Этот индекс представляет собой просто целое число: 0 для первого элемента, 1 для второго и т.д. Это означает, что члены массива можно перебрать в цикле:

```
int i;
for (i = 0; i < 3; i++)
{
  Console.WriteLine("Имя с индексом {0} - {1}", i, friendNames[i]);
}
```

Массивы имеют единственный *базовый тип*, т.е. все отдельные члены массива относятся к одному и тому же типу. Например, базовым типом приведенного здесь массива `friendNames` является `string`, т.к. этот массив предназначен для хранения переменных типа `string`. Члены массивов обычно называют *элементами*.

Объявление массивов

Массивы объявляются следующим образом:

```
<базовыйТип>[] <имя>;
```

Здесь *<базовыйТип>* может быть любым типом переменных, в том числе и типом перечисления и структуры. Перед обращением к массивам их необходимо инициализировать. Нельзя сразу обращаться к элементам массива или присваивать им значения, как в следующем коде:

```
int[] myIntArray;
myIntArray[10] = 5;
```

Инициализировать массивы можно двумя способами: либо указать все содержимое массива с помощью литералов, либо указать размер массива и использовать ключевое слово `new` для инициализации всех его элементов.

Для задания массива с помощью литеральных значений нужно просто записать заключенный в фигурные скобки список значений элементов, разделенных запятыми:

```
int[] myIntArray = {5, 9, 10, 2, 99};
```

Здесь определяется массив `myIntArray`, состоящий из пяти элементов, каждому из которых присваивается целочисленное значение.

Второй подход использует следующий синтаксис:

```
int[] myIntArray = new int[5];
```

Здесь ключевое слово `new` используется для явной инициализации массива, а константное значение в квадратных скобках — для определения его размера. При таком подходе всем членам массива присваиваются стандартные значения; для числовых типов это 0. В такой инициализации можно применять и переменные с неконстантными значениями:

```
int[] myIntArray = new int[arraySize];
```

При желании оба подхода можно комбинировать:

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

Однако при этом размеры массива и списка значений должны совпадать. То есть следующая строка ошибочна:

```
int[] myIntArray = new int[10] {5, 9, 10, 2, 99};
```

Здесь массив определен как состоящий из 10 членов, но инициализируются только 5 членов, поэтому компилятор сообщит об ошибке. Кстати, если при таком подходе размер определяется с помощью переменной, то эта переменная должна быть обязательно константной:

```
const int arraySize = 5;
int[] myIntArray = new int[arraySize] {5, 9, 10, 2, 99};
```

При отсутствии ключевого слова `const` код компилироваться не будет.

Как и другие типы переменных, инициализировать массив в той же строке, в которой он объявляется, не обязательно. Можно сделать, например, и так:

```
int[] myIntArray;
myIntArray = new int[5];
```

В следующем практическом занятии демонстрируется создание массива строк для хранения имен и работа с ним.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование массива

1. Создайте новое консольное приложение с именем `Ch05Ex04` и сохраните его в каталоге `C:\BegVCSharp\Chapter05`.
2. Добавьте в файл `Program.cs` следующий код:

```
static void Main(string[] args)
{
   string[] friendNames = {"Robert Barwell", "Mike Parry", "Jeremy Beacock"};
   int i;
   Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
                // Вот {0} моих друзей
   for (i = 0; i < friendNames.Length; i++)
   {
      Console.WriteLine(friendNames[i]);
   }
   Console.ReadKey();
}
```

Фрагмент кода ***Ch05Ex04\Program.cs***

3. Запустите приложение. На рис. 5.9 показан результат, который должен получиться.

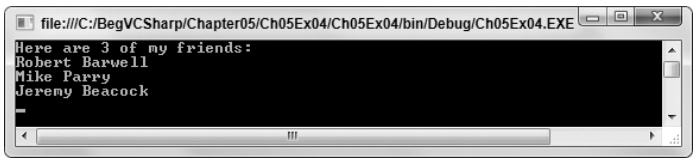


Рис. 5.9. *Приложение* `Ch05Ex04` *в действии*

Описание работы

В приведенном коде сначала создается массив `string` с тремя значениями, которые затем выводятся в цикле `for`. Обратите внимание, что выражение `friendNames.Length` обеспечивает доступ к информации о количестве элементов в массиве:

```
Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
```

Это удобный способ получения размера массива. При выводе значений массива в цикле `for` можно легко ошибиться. Например, попробуйте изменить < на <=:

```
for (i = 0; i <= friendNames.Length; i++)
{
   Console.WriteLine(friendNames[i]);
}
```

После компиляции этого кода появится диалоговое окно, показанное на рис. 5.10.

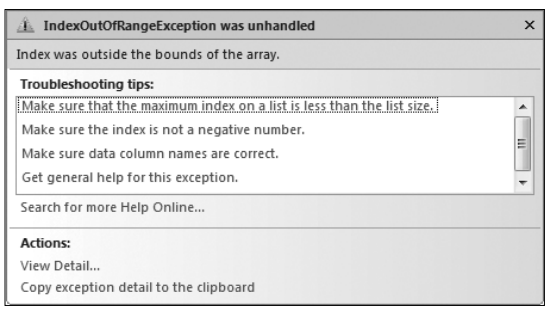


Рис. 5.10. *Диалоговое окно с сообщением об ошибке выхода индекса за допустимые пределы*

На этом рисунке видно, что обнаружена попытка обращения к элементу `friendNames[3]`. Поскольку нумерация индексов в массиве начинается с 0, то в данном случае последним элементом является `friendNames[2]`. При попытке обращения к элементам за границей массива возникает ошибка. Однако существует еще один более гибкий способ обращения к членам массива — это использование циклов `foreach`.

Циклы `foreach`

Цикл `foreach` позволяет обращаться к каждому элементу в массиве с помощью такого простого синтаксиса:

```
foreach ( <базовыйТип> <имя> in <массив> )
{
   // Можно использовать <имя> для каждого элемента
}
```

Этот цикл перебирает все элементы массива и помещает их поочередно в переменную *<имя>* без всякого риска обращения к несуществующим элементам. Такой подход позволяет не беспокоиться о количестве элементов в массиве и быть уверенным, что каждый из них будет использован в цикле. С его помощью код из предыдущего примера можно изменить следующим образом:

```
static void Main(string[] args)
{
   string[] friendNames = {"Robert Barwell", "Mike Parry", "Jeremy Beacock"};
   Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
   foreach (string friendName in friendNames)
   {
      Console.WriteLine(friendName);
   }
   Console.ReadKey();
}
```

Результат выполнения этого кода в точности совпадает с результатом работы кода из последнего практического занятия. Главное различие между циклами `foreach` и стандартными циклами `for` состоит в том, что `foreach` предоставляет *доступ только для чтения* содержимого массива, что не позволяет изменять значения элементов. То есть невозможно выполнить, к примеру, следующее действие:

```
foreach (string friendName in friendNames)
{
   friendName = "Мишка косолапый";
}
```

Компилятор просто не пропустит такой код. Но при использовании обычного цикла `for` присваивание значений элементам массива возможно.

Многомерные массивы

Многомерным называется такой массив, в котором для получения доступа к элементам применяется несколько индексов. Например, предположим, что требуется нарисовать карту высот холма по замеренным позициям. Позицию можно задать двумя координатами — `x` и `y`. Эти две координаты можно использовать в качестве индексов, чтобы хранить в массиве с именем `hillHeight` значения высот, соответствующих каждой паре координат. В такой ситуации может помочь многомерный массив.

Двумерные массивы объявляются следующим образом:

```
<базовыйТип>[,] <имя>;
```

Для объявления массивов с большим количеством измерений нужно указать больше запятых:

```
<базовыйТип>[,,,] <имя>;
```

Здесь объявлен четырехмерный массив. Для присваивания значений применяется похожий синтаксис, в котором запятые служат для разделения размерностей. Объявить и инициализировать двухмерный массив `hillHeight` с базовым типом `double` и размерностью 3 по координате `x`, и 4 по координате `y` можно следующим образом:

```
double[,] hillHeight = new double[3,4];
```

Но, конечно, для начального присваивания можно использовать и литеральные значения. Эти значения должны иметь вид вложенных блоков, заключенных в фигурные скобки, и отделяться друг от друга запятыми:

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
```

Данный массив имеет такие же измерения, что и предыдущий, т.е. три строки и четыре столбца. Литеральные значения определяют эти измерения неявным образом.

Для доступа к отдельным элементам многомерного массива нужно указать отделенные запятой индексы:

```
hillHeight[2,1]
```

Теперь эти элементы можно обрабатывать так же, как и все остальные. Приведенное выражение означает обращение ко второму элементу третьего вложенного массива, который был определен ранее (т.е. к значению 4). Напомним, что отсчет начинается с 0, а первое число представляет номер вложенного массива. Другими словами, первое число указывает пару фигурных скобок, а второе — элемент внутри этой пары скобок. Графически этот массив можно представить так, как показано на рис. 5.11.

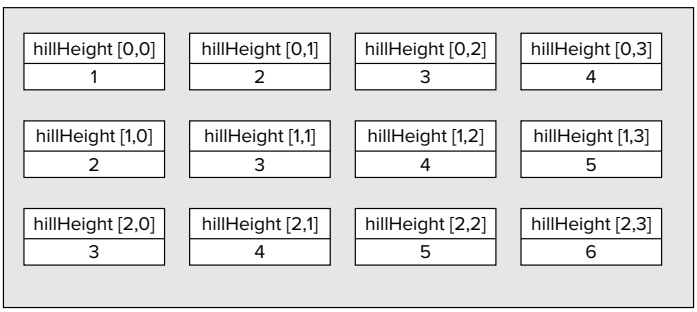


Рис. 5.11. *Графическое представление массива* `hillHight`

Цикл `foreach` предоставляет доступ ко всем элементам в многомерном массиве точно так же, как и в одномерных массивах:

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
foreach (double height in hillHeight)
{
  Console.WriteLine("{0}", height);
}
```

Порядок, в котором элементы выводятся, совпадает с тем, что использовался для присваивания литеральных значений:

```
hillHeight[0,0]
hillHeight[0,1]
hillHeight[0,2]
hillHeight[0,3]
hillHeight[1,0]
hillHeight[1,1]
hillHeight[1,2]
...
```

Массивы массивов

Многомерные массивы, о которых рассказывалось в предыдущем разделе, называются *прямоугольными*, потому что все "строки" в них имеют одинаковый размер. Например, в последнем рассмотренном случае любой из возможных координат `x` могла соответствовать координата `y` от 0 до 3.

Однако массивы могут быть и *зубчатыми*, когда "строки" имеют разные размеры. Для этого нужен массив, каждый элемент которого также является массивом. При желании можно создавать и массивы массивов, состоящие из массивов, или даже еще более сложные конструкции. Однако все эти варианты возможны только для массивов одного и того же базового типа.

В объявлениях массивов, состоящих из массивов, указываются нескольких пар квадратных скобок:

```
int[][] jaggedIntArray;
```

К сожалению, инициализация подобных массив не так проста, как инициализация многомерных массивов. Например, за предыдущим объявлением нельзя записать такую строку кода:

```
jaggedIntArray = new int[3][4];
```

Даже если бы это было возможно, данная строка не была бы особенно полезной, т.к. того же эффекта для многомерных массивов можно добиться и с гораздо меньшими усилиями. Нельзя написать и так:

```
jaggedIntArray = {{1, 2, 3}, {1}, {1, 2}};
```

Существуют всего два возможных варианта. Первый — инициализировать сначала массив, содержащий другие массивы (которые далее для краткости будут называться просто подмассивами), а затем по очереди инициализировать подмассивы:

```
jaggedIntArray = new int[2][];
jaggedIntArray[0] = new int[3];
jaggedIntArray[1] = new int[4];
```

Второй вариант — использовать модифицированную версию приведенного выше литерального присваивания:

```
jaggedIntArray = new int[3][] {new int[] {1, 2, 3},
                               new int[] {1},
                               new int[] {1, 2}   };
```

Инициализировать массив можно и в той же строке, в которой он объявляется:

```
int[][] jaggedIntArray = {new int[] {1, 2, 3}, new int[] {1}, new int[] {1, 2}};
```

С зубчатыми массивами можно использовать циклы `foreach`, но обычно их делают вложенными, чтобы получить фактические данные. Например, предположим, что имеется показанный ниже зубчатый массив, содержащий 10 других массивов, каждый из которых, в свою очередь, тоже содержит массив целых чисел, представляющих собой делители целых чисел от 1 до 10:

```
int[][] divisors1To10 = { new int[] {1},
                          new int[] {1, 2},
                          new int[] {1, 3},
                          new int[] {1, 2, 4},
                          new int[] {1, 5},
                          new int[] {1, 2, 3, 6},
                          new int[] {1, 7},
                          new int[] {1, 2, 4, 8},
                          new int[] {1, 3, 9},
                          new int[] {1, 2, 5, 10} };
```

Выполнение следующего кода завершится ошибкой:

```
foreach (int divisor in divisors1To10)
{
  Console.WriteLine(divisor);
}
```

Объясняется это тем, что массив `divisors1To10` содержит элементы типа `int[]`, а не `int`. Поэтому в цикле нужно перебирать элементы каждого подмассива, а также всего массива:

```
foreach (int[] divisorsOfInt in divisors1To10)
{
  foreach(int divisor in divisorsOfInt)
  {
    Console.WriteLine(divisor);
  }
}
```

Как видно, использование зубчатых массивов может сильно усложнить код приложения. Из-за этого в большинстве случаев проще использовать прямоугольные массивы или какой-то другой более простой метод хранения данных. Однако все же могут встречаться ситуации, в которых без применения таких массивов не обойтись, и тогда умение работы с ними совсем не повредит.

Обработка строк

До сих пор наши способы применения строк сводились к выводу строк на консоль, считыванию строк с консоли и конкатенации строк с помощью операции +. В процессе программирования более интересных приложений быстро становится понятно, что обработка строк выполняется очень часто. Так что стоит потратить несколько страниц на изучение некоторых наиболее распространенных приемов обработки строк, которые доступны в языке C#.

Для начала отметим, что переменную типа `string` можно рассматривать как доступный только для чтения массив переменных `char`. Это означает, что отдельные символы можно выбирать с помощью приблизительно такого синтаксиса:

```
string myString = "Строка";
char myChar = myString[1];
```

Однако присваивать отдельные символы подобным образом нельзя. Для получения массива `char`, который допускает запись, можно применять показанный ниже код. В нем применяется команда `ToCharArray()`:

```
string myString = "Строка";
char[] myChars = myString.ToCharArray();
```

После этого с массивом `char` можно работать обычным образом. Кроме того, строки можно использовать в циклах `foreach`:

```
foreach (char character in myString)
{
  Console.WriteLine("{0}", character);
}
```

Как и в случае массивов, получить количество элементов в строке можно с помощью выражения `myString.Length`:

```
string myString = Console.ReadLine();
Console.WriteLine("Введено {0} символов.", myString.Length);
```

В других основных приемах обработки строк используются команды наподобие *<строка>*.ToCharArray(). Две простых, но полезных команды — *<строка>*.ToLower() и *<строка>*.ToUpper(). Эти команды позволяют преобразовывать строки соответственно в нижний и в верхний регистр. Чтобы понять, что в этом полезного, предположим, что требуется проверить ввод конкретного ответа от пользователя — например, строки yes. Преобразование введенной пользователем строки в нижний регистр позволит проверять наличие и таких строк, как YES, Yes, yeS и т.д. — подобный пример уже приводился в предыдущей главе.

```
string userResponse = Console.ReadLine();
if (userResponse.ToLower() == "yes")
{
   // Ответное действие.
}
```

Эта команда, как и прочие команды в данном разделе, на самом деле не изменяет строку, к которой применяется. Вместо этого применение данной команды приводит к созданию новой строки, которую можно либо сравнить с еще какой-нибудь строкой (как это сделано здесь), либо присвоить другой переменной. В роли этой другой переменной может выступать и та же самая обрабатываемая переменная:

```
userResponse = userResponse.ToLower();
```

Не забывайте про это, поскольку использование просто такой строки кода:

```
userResponse.ToLower();
```

на самом деле не даст никакого результата.

Существуют и другие средства, которые могут упростить интерпретацию вводимых пользователем данных. Что если пользователь случайно введет в начале или в конце входных данных лишний пробел? В таком случае приведенный выше код работать не будет. Нужно сначала удалить концевые пробелы, что можно сделать с помощью команды *<строка>*.Trim():

```
string userResponse = Console.ReadLine();
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
{
   // Ответное действие.
}
```

Данный код распознает и такие строки, как " YES" и "Yes ".

Эти команды позволяют удалять и любые другие символы, если указать их в массиве char, например:

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = Console.ReadLine();
userResponse = userResponse.ToLower();
userResponse = userResponse.Trim(trimChars);
if (userResponse == "y")
{
   // Ответное действие.
}
```

Приведенный код удаляет все пробелы, буквы e и буквы s в начале и конце строки. При отсутствии любых других символов в строке он распознает такие строки, как "Yeeeees", " y" и т.д.

Кроме того, имеются команды *<строка>*.TrimStart() и *<строка>*.TrimEnd(), которые удаляют пробелы, соответственно, только в начале и только в конце строки. Для них также можно задавать массивы char.

Существуют еще две других строковых команды, предназначенные для работы с пробелами в строках: *<строка>*.PadLeft() и *<строка>*.PadRight(). Они добавляют пробелы слева и справа от строки, дополняя ее до указанной длины. Применяются они следующим образом:

```
<строка>.PadX(<нужнаяДлина>);
```

Вот, например:

```
myString = "Aligned";
myString = myString.PadLeft(10);
```

Выполнение этого кода приведет к добавлению слева от слова Aligned в myString трех пробелов. Данные команды очень удобны для выравнивания строк в столбцах, особенно при позиционировании строк, содержащих числа.

Как и в случае команд усечения, эти команды тоже можно использовать другим способом, указав символ, который должен применяться для дополнения строки. Указывать можно только один символ char, а не массив символов, как в случае усечения:

```
myString = "Aligned";
myString = myString.PadLeft(10, '-');
```

Выполнение этого кода приведет к добавлению в начале myString трех символов -.

Существует еще очень много подобных команд для обработки строк, но многие из них могут понадобиться только в очень специфических ситуациях. Поэтому они будут рассматриваться по мере их применения в последующих главах. Но прежде чем продолжить, стоит рассмотреть еще одну возможность, доступную как в Visual C# 2010 Express Edition, так и в Visual Studio 2010. Вы уже могли обратить на нее внимание в предыдущих главах и особенно в этой. В следующем практическом занятии демонстрируется функция автоматического завершения команд (auto-completion), посредством которой IDE-среда пытается помогать разработчику, предлагая различные варианты кода, которые могут понадобиться в конкретной ситуации.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Автоматическое завершение операторов в VS

1. Создайте новое консольное приложение с именем Ch05Ex05 и сохраните его в каталоге C:\BegVCSharp\Chapter05.
2. Добавьте в файл Program.cs следующий код, в точности набирая указанные символы и обращая внимание на всплывающие окна, которые будут появляться при вводе:

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.
}
```

Фрагмент кода ***Ch05Ex05\Program.cs***

Рис. 5.12. *Окно, появившееся после ввода последней строки*

3. После ввода последней точки должно появиться окно, показанное на рис. 5.12.
4. Не перемещая курсор, введите **sp**. Всплывающее окно изменит вид, и появится подсказка желтого цвета, показанная на рис. 5.13.

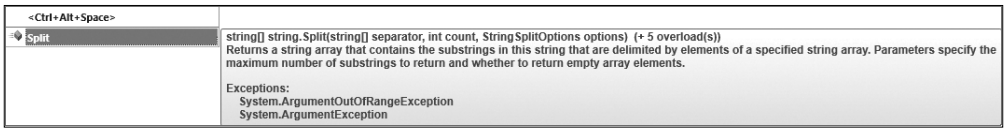


Рис. 5.13. *Всплывающая подсказка*

5. Введите символы **(se**. Появится другое всплывающее окно, показанное на рис. 5.14.

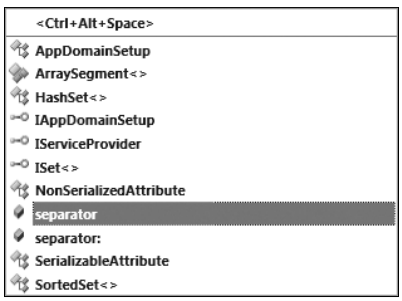


Рис. 5.14. *Окно, появившееся после ввода символов "(se"*

6. После этого введите символы **);**. Окно пропадет, а код будет выглядеть так:

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.Split(separator);
}
```

7. Добавьте следующий код, обращая внимание на окна, которые будут появляться по мере его ввода:

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.Split(separator);
   foreach (string word in myWords)
   {
      Console.WriteLine("{0}", word);
   }
   Console.ReadKey();
}
```

8. Запустите приложение. На рис. 5.15 показан результат, который должен получиться.

Рис. 5.15. *Приложение* `Ch05Ex05` *в действии*

Описание работы

В приведенном коде важны два момента: использование новой строковой команды и применение функции автоматического завершения кода. Новая строковая команда — `<строка>.Split()` — преобразует строку в массив строк, разбивая ее на части по указанным символам. Эти символы указываются в массиве `char`, который в нашем случае содержит единственный элемент — символ пробела:

```
char[] separator = {' '};
```

Следующий код формирует подстроки, разбивая исходную строку по каждому пробелу, и сохраняет их в массиве отдельных слов:

```
string[] myWords;
myWords = myString.Split(separator);
```

Далее все слова в этом массиве выводятся на консоль с помощью цикла `foreach`:

```
foreach (string word in myWords)
{
  Console.WriteLine("{0}", word);
}
```

Каждое полученное слово не содержит пробелов ни внутри, ни в конце. Все они удаляются в результате работы команды `Split()`.

А теперь вернемся к функции автоматического завершения кода. VS и VCE — весьма интеллектуальные среды, которые способны извлекать много информации о коде по мере его ввода. Даже при вводе первого символа в новой строке IDE-среда уже пытается помочь, предлагая различные варианты ключевых слов, имен переменных, имен типов и т.д. После ввода в приведенном коде всего лишь трех букв `str` среда правильно предположила, что требуется ввести слово `string`. Эта функция еще более полезна при вводе имен переменных. При наборе длинных фрагментов кода имена переменных часто забываются. Но поскольку по мере ввода IDE-среда отображает всплывающий список всех возможных имен, имена нужных переменных нетрудно отыскать, не возвращаясь к их описаниям.

К моменту ввода после `myString` символа точки IDE-среда уже знает, что `myString` является строкой, понимает, что требуется указать строковую команду, и предлагает список возможных вариантов. На этом этапе можно прекратить ввод и просто выбрать в этом списке желаемую команду с помощью клавиш со стрелками вверх и вниз. При просмотре доступных вариантов IDE-среда выводит описание выделенной команды и ее синтаксиса.

При продолжении ввода символов IDE-среда автоматически перемещает выбранную команду наверх списка наиболее вероятных команд. Когда будет показана нужная команда, можно продолжать ввод так, будто бы уже введено полное имя. То есть после ввода открывающей круглой скобки можно сразу же указывать необходимую некоторым командам дополнительную информацию — а IDE-среда даже подскажет, в каком формате должна вводиться эта информация, отображая возможные варианты для команд, которые принимают различное количество параметров.

Эта функция IDE-среда (называемая IntelliSense) чрезвычайно удобна и позволяет легко находить сведения о любых незнакомых типах. Если интересно, можете с ее помощью, например, просмотреть все команды, которые поддерживает тип `string`, и поэкспериментировать с ними.

Иногда такая отображаемая информация может закрывать уже введенный код, а это не всем и не всегда нравится. В загороженном коде может содержаться нужная в данный момент информация. Нажатие клавиши <Ctrl> сделает список команд прозрачным и позволит видеть скрытый текст.

Резюме

Благодаря настоящей главе, вы расширили знания о переменных. Пожалуй, наиболее важной из рассмотренных в этой главе тем — это преобразования типов, которые будут встречаться в книге постоянно. Уверенное освоение изложенных концепций существенно облегчит понимание дальнейшего материала.

Помимо этого в главе рассмотрены еще несколько типов переменных, позволяющих более удобно хранить данные. Перечисления делают код понятнее с помощью наглядных значений, структуры объединяют несколько связанных между собой элементов данных в одном месте, а массивы группируют однотипные данные. Все эти типы будут постоянно применяться в оставшейся части книги.

И в завершение главы было рассказано об обработке строк, о базовых приемах и принципах. Для этого имеется много строковых команд; здесь было рассказано лишь о некоторых из них, но вы уже знаете, как просмотреть список доступных команд в IDE-среде. Не бойтесь экспериментировать. По крайней мере, одно из следующих упражнений можно выполнить с использованием одной или более строковых команд, которые в главе не рассматривались, и вам придется обнаружить их самостоятельно.

В главе были рассмотрены следующие темы, связанные с переменными:

- преобразования типов;
- перечисления;
- структуры;
- массивы;
- обработка строк.

Упражнения

1. Какие из следующих операций преобразования не могут выполняться неявно:

   а) `int` в `short`
   б) `short` в `int`
   в) `bool` в `string`
   г) `byte` в `float`

2. Создайте на базе типа `short` код для перечисления `color`, содержащего все цвета радуги, а также черный и белый цвета. Может ли такое перечисление основываться на типе `byte`?

3. Измените пример с генератором множества Мандельброта из предыдущей главы так, чтобы в нем использовалась следующая структура для комплексных чисел:

```
struct imagNum
{
  public double real, imag;
}
```

4. Будет ли компилироваться данный код?

```
string[] blab = new string[5]
string[5] = 5th string.
```

   Обоснуйте ответ.

5. Напишите консольное приложение, которое запрашивает у пользователя строку и выводит все ее символы в обратном порядке.

6. Напишите консольное приложение, которое запрашивает строку и заменять в ней все вхождения слова `no` словом `yes`.
7. Напишите консольное приложение, которое заключает каждое слово в строке в двойные кавычки.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Преобразование типов	Значения можно преобразовывать из одного типа в другой, но эти преобразования должны подчиняться определенным правилам. Неявные преобразования выполняются автоматически, но только если все возможные значения исходного типа доступны в целевом типе. Можно указывать и явные преобразования, но тогда присваивания значений могут дать неожиданный результат или привести к ошибкам.
Перечисления	Перечисления — это типы с дискретным множеством значений, каждое из которых имеет имя. Перечисления определяются с помощью ключевого слова `enum`. Это повышает читабельность кода, т.к. имена членов перечисления обычно поясняют их назначение. Перечисления основываются на числовом типе (по умолчанию `int`), и это свойство позволяет преобразовывать значения перечислений в числа и обратно или идентифицировать эти значения.
Структуры	Структуры — это типы, содержащие сразу несколько различных значений. Они определяются с помощью ключевого слова `struct`. Все значения, которые хранятся в структуре, имеют имя и тип. Значения, хранящиеся в структуре, совсем не обязательно должны быть одного типа.
Массивы	Массив — это коллекция значений одного типа. Массивы имеют фиксированный размер (длину), который определяет, сколько значений может содержать массив. Можно определять многомерные или зубчатые массивы для хранения различных объемов данных. Перебирать элементы массива можно с помощью цикла `foreach`.

6

Функции

В ЭТОЙ ГЛАВЕ...

- Определение и использование простых функций, не возвращающих данные
- Передача данных в функции и из функций
- Области видимости переменных
- Использование аргументов командной строки с помощью функции `Main()`
- Передача функций в качестве членов структурных типов
- Перегрузка функций
- Использование делегатов

Весь приведенный до сих пор в книге код был оформлен в виде одного блока, иногда с циклами для повторения строк кода и условными разветвлениями. Для выполнения какого-то действия с данными весь необходимый код размещался там, где он должен работать.

Такая структура кода не всегда удобна. Довольно часто нужно, чтобы некоторые задачи, вроде поиска максимального значения в массиве, выполнялись в нескольких местах программы. Конечно, можно добавлять в приложение одинаковые (или почти одинаковые) разделы кода всякий раз, когда они нужны, но такой подход имеет свои недостатки. Изменение даже какой-то одной мелочи, касающейся общей задачи (например, исправление ошибки в коде), в таком случае означает, что соответствующие изменения потребуется вносить в несколько разделов кода, которые могут быть разбросаны по всему приложению. Пропуск одного из этих разделов может привести к серьезным последствиям и сделать неработоспособным все приложение. К тому же это существенно удлиняет приложение.

Решить подобную проблему позволяют *функции*. Функции в языке C# являются средствами оформления блоков кода, которые могут выполняться из любого места приложения.

Функции того вида, который изучается в настоящей главе, называются методами, но в программировании для .NET этот термин имеет очень специфическое значение, которое будет объяснено позже. Поэтому пока данный термин употребляться не будет.

Например, можно создать функцию, вычисляющую максимальное значение в массиве. Ее можно применять из любого места кода и в каждом случае использовать для этого одинаковые строки кода. Поскольку записать этот код нужно только один раз, любые вносимые в него изменения будут отражаться на действии функции при всяком ее применении. То есть функция содержит *многократно используемый* код.

Еще одним преимуществом функций является то, что они делают код понятнее, поскольку могут применяться для группирования взаимосвязанного кода. Благодаря этому тело приложения сокращается за счет выделения внутренних действий во внешние блоки. Это похоже на сворачивание разделов кода в IDE-среды с помощью структурного представления, при этом структура приложения становится более логичной.

Кроме того, функции позволяют создавать универсальный код для выполнения одних и тех же действий с различными данными. Необходимую для работы функции информацию можно передать в виде параметров, а результаты их работы — в виде возвращаемых значений. В приведенном выше примере в качестве параметра можно передавать массив, а в качестве возвращаемого значения — максимальную величину в этом массиве. Это означает, что одну и ту же функцию можно использовать для работы с различными массивами. Имя и параметры функции (но не тип возвращаемого ей значения) вместе называются *сигнатурой* функции.

В конце главы будут рассмотрены две более сложные темы — *перегрузка функций* и *делегаты*. Перегрузкой функций называется прием, позволяющий предоставлять несколько функций с одинаковым именем, но разными параметрами, а делегат — это тип переменной, позволяющий использовать функции косвенным образом. Делегат может применяться для вызова любой функции, возвращаемый тип и параметры которой соответствуют определенным в нем, а это позволяет делать во время выполнения выбор между несколькими разными функциями.

Определение и использование функций

В этом разделе вы узнаете, как добавлять функции в приложения и затем использовать (вызывать) их из кода. Вначале демонстрируются простые функции, которые не обмениваются данными с вызывающим кодом, а затем более сложные функции и способы работы с ними. Начнем сразу с практического занятия.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Определение и использование простой функции

1. Создайте новое консольное приложение с именем `Ch06Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter06`.
2. Добавьте в файл `Program.cs` следующий код:

```
class Program
{
   static void Write()
   {
      Console.WriteLine("Text output from function.");
                        // Текст, сгенерированный функцией
   }
   static void Main(string[] args)
   {
      Write();
      Console.ReadKey();
   }
}
```

Фрагмент кода ***Ch06Ex01\Program.cs***

3. Запустите приложение. На рис. 6.1 показан результат, который должен получиться.

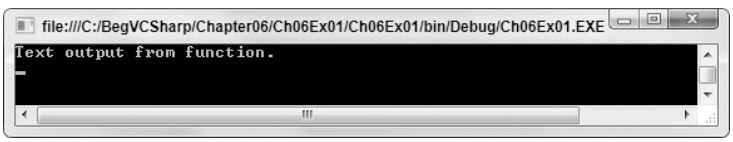


Рис. 6.1. *Приложение* `Ch06Ex01` *в действии*

Описание работы

В следующих четырех строках кода определяется функция с именем `Write()`:

```
static void Write()
{
   Console.WriteLine("Text output from function.");
}
```

Код данной функции просто выводит в окно консоли некоторый текст, но важно пока не это, а механизм определения и использования функций.

Определение функции состоит из следующих элементов:

- два ключевых слова: `static` и `void`;
- имя функции, за которым следуют круглые скобки — `Write()`;
- заключенный в фигурные скобки блок кода, который подлежит выполнению.

Имена функций обычно пишутся в стиле языка Pascal.

Код, определяющий функцию `Write()`, очень похож на другой код в приложении:

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

Это объясняется тем, что весь код, который нам приходилось писать до сих пор (кроме определений типов), был частью функции — а именно, `Main()` — которая является так называемой *точкой входа* для консольного приложения. При выполнении любого C#-приложения первой вызывается функция точки входа, а при завершении выполнения этой функции завершается и работа всего приложения. Весь исполняемый код на C# должен обязательно иметь такую точку входа.

Единственным отличием между функциями `Main()` и `Write()` (кроме содержащихся в них строк) является текст внутри круглых скобок после имени функции `Main`. С помощью этого текста задаются параметры, о чем более подробно будет рассказано чуть позже.

Как уже было сказано, и `Main()`, и `Write()` определены с ключевыми словами `static` и `void`. Ключевое слово `static` относится к концепциям объектно-ориентированного программирования и поэтому будет рассмотрено позже. Пока необходимо запомнить лишь то, что оно должно обязательно присутствовать во всех функциях, которые предлагается создавать в приложениях, описываемых в настоящем разделе.

А вот смысл ключевого слова `void` объяснить проще. Оно служит для указания, что функция не возвращает никакого значения. Ниже будет описано, как указать, что функция имеет возвращаемое значение.

Далее идет код, вызывающий функцию:

```
Write();
```

Здесь просто записано имя функции, а за ним — пустые круглые скобки. Когда во время выполнения программы поток управления дойдет до этого места, будет выполнен код, содержащийся внутри функции `Write()`.

Круглые скобки, как в определении, так и в вызове функции, являются обязательными. Если их удалить, код не будет компилироваться.

Возвращаемые значения

Наиболее простой способ для обмена данными с функцией — использование возвращаемого значения. Функции, которые возвращают значения, *вычисляют* эти значения, точно так же, как вычисляются значения переменных в выражениях. И как и переменные, возвращаемые значения имеют тип.

Например, можно написать функцию `GetString()` с возвращаемым значением типа `string`. Использовать такую функцию в коде можно следующим образом:

```
string myString;
myString = GetString();
```

Можно также создать функцию `GetVal()` с возвращаемым значением типа `double`. Такая функция может участвовать в математическом выражении:

```
double myVal;
double multiplier = 5.3;
myVal = GetVal() * multiplier;
```

Чтобы функция возвращала значение, в нее необходимо внести два изменения:

- в объявлении функции вместо ключевого слова `void` нужно указать тип возвращаемого значения;
- в конце выполнения функции нужно передать возвращаемое значение в вызывающий код с помощью ключевого слова `return`.

В коде все это выглядит так, как вы уже не раз видели в функциях консольных приложений:

```
static <возвращаемыйТип> <имяФункции> ()
{
   ...
   return <возвращаемоеЗначение>;
}
```

Единственное ограничение — тип выражения <*возвращаемоеЗначение*> должен либо совпадать с типом <*возвращаемыйТип*>, либо неявно преобразовываться в этот тип. Однако сам <*возвращаемыйТип*> может быть каким угодно, включая и составные типы. Вот очень простой пример:

```
static double GetVal()
{
   return 3.2;
}
```

Конечно, обычно возвращаемые значения являются результатом какой-то обработки, выполняемой внутри функции, а в данном примере результат проще было бы получить с помощью переменной `const`.

При достижении оператора `return` поток управления программы сразу же возвращается в вызывающий код. Никакие строки кода функции после этого оператора не выполняются, хотя это вовсе не означает, что операторы `return` можно размещать только в последней строке тела функции. Оператор `return` можно использовать и раньше, например, после какого-то разветвления. Он может находиться в цикле `for`, блоке `if` или любой другой структуре и все равно приведет к немедленному выходу из структуры и завершению выполнения функции, например:

```
static double GetVal()
{
   double checkVal;
   // CheckVal присваивается некоторое значение (здесь не показано, как).
   if (checkVal < 5)
      return 4.7;
   return 3.2;
}
```

Здесь в зависимости от значения `checkVal` может быть возвращено одно из двух значений. Единственным ограничением в данном случае является то, что оператор `return` должен быть выполнен до достижения фигурной скобки, закрывающей функцию. Следующий код недопустим:

```
static double GetVal()
{
   double checkVal;
   // CheckVal присваивается некоторое значение.
   if (checkVal < 5)
      return 4.7;
}
```

Если `checkVal >= 5`, ни одного оператора `return` не выполняется, что недопустимо. Все пути обработки должны обязательно завершаться оператором `return`. В большинстве случаев компилятор распознает эту ошибку и выдает сообщение `Not all code paths return a value` (Не все пути кода возвращают значение).

И, наконец, последнее: оператор `return` может применяться и в функциях, объявленных с ключевым словом `void` (т.е. которые не имеют возвращаемого значения). Такие функции будут просто завершаться. При таком использовании оператора `return` указание возвращаемого значения между ключевым словом `return` и последующей точкой с запятой считается ошибкой.

Параметры

Когда функция должна принимать параметры, необходимо указывать следующее:

- список принимаемых функцией параметров вместе с их типами в определении этой функции;
- соответствующий список параметров в каждом вызове функции.

Это означает применение показанного ниже кода, в котором может присутствовать любое количество параметров, при этом для каждого из них должны быть указаны тип и имя:

```
static <возвращаемыйТип> <имяФункции> ( <типПараметра> <имяПараметра>, ...)
{
   ...
   return <возвращаемоеЗначение>;
}
```

Параметры отделяются друг от друга запятыми, и все они доступны в коде функции как обычные переменные. Например, вот простая функция, которая принимает два параметра `double` и возвращает их произведение:

```
static double Product(double param1, double param2)
{
   return param1 * param2;
}
```

Более сложный пример приведен в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Обмен данными с функцией (часть 1)

1. Создайте новое консольное приложение с именем `Ch06Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter06`.
2. Добавьте в файл `Program.cs` следующий код:

```
class Program
{
   static int MaxValue(int[] intArray)
   {
      int maxVal = intArray[0];
      for (int i = 1; i < intArray.Length; i++)
      {
         if (intArray[i] > maxVal)
            maxVal = intArray[i];
      }
      return maxVal;
   }
   static void Main(string[] args)
   {
      int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
      int maxVal = MaxValue(myArray);
      Console.WriteLine("The maximum value in myArray is {0}", maxVal);
                        // Максимальное значение myArray равно
      Console.ReadKey();
   }
}
```

Фрагмент кода ***Ch06Ex02\Program.cs***

3. Запустите приложение. На рис. 6.2 показан результат, который должен получиться.

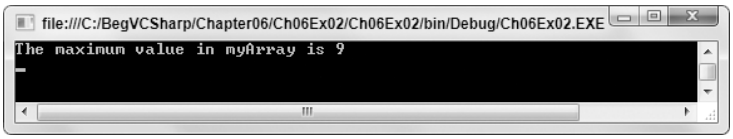


Рис. 6.2. *Приложение* `Ch06Ex02` *в действии*

Описание работы

В приведенном коде содержится функция, которая делает то, о чем шла речь в начале главы — принимает в качестве параметра массив целых чисел и возвращает максимальное из них. Вот определение этой функции:

```
static int MaxValue(int[] intArray)
{
  int maxVal = intArray[0];
  for (int i = 1; i < intArray.Length; i++)
  {
    if (intArray[i] > maxVal)
      maxVal = intArray[i];
  }
  return maxVal;
}
```

Итак, эта функция называется `MaxValue()` и имеет один определенный параметр — массив типа `int` по имени `intArray`, а также возвращаемый тип `int`. Максимальное значение вычисляется очень просто. Сначала в локальную переменную `maxVal` заносится первое значение из массива, после чего значение этой переменной сравнивается с каждым из остальных элементов массива. Если в каком-то из элементов содержится большее значение, чем в `maxVal`, оно становится новым текущим значением `maxVal`. По завершении цикла `maxVal` содержит максимальное значение из имеющихся в массиве и возвращается с помощью оператора `return`.

В коде функции `Main()` объявляется и инициализируется простой массив целых чисел, который используется функцией `MaxValue()`:

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
```

Вызов `MaxValue()` присваивает значение целочисленной переменной `maxVal`:

```
int maxVal = MaxValue(myArray);
```

Далее это значение выводится на экран с помощью вызова `Console.WriteLine()`:

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```

Соответствие параметров

При вызове функции параметры должны указываться точно так же, как и в определении функции, т.е. должны соответствовать типы, количество и порядок следования параметров. Например, функцию

```
static void MyFunction(string myString, double myDouble)
{
  ...
}
```

нельзя вызывать так:

```
MyFunction(2.6, "Hello");
```

Здесь в качестве первого параметра передается значение double, а в качестве второго — значение string, что не соответствует порядку, в котором эти параметры были указаны в определении функции.

Эту функцию также нельзя вызывать и следующим образом:

```
MyFunction("Hello");
```

Здесь передается только один параметр string, а нужно два. При попытке вызвать функцию любым из двух приведенных выше способов компилятор сообщит об ошибке из-за несоответствия сигнатуре.

Как уже упоминалось, под сигнатурой подразумевается имя и параметры функции.

Рассмотренная выше функция MaxValue() может применяться только для получения максимального целого значения в массиве значений int. Если изменить код Main() следующим образом:

```
static void Main(string[] args)
{
   double[] myArray = {1.3, 8.9, 3.3, 6.5, 2.7, 5.3};
   double maxVal = MaxValue(myArray);
   Console.WriteLine("The maximum value in myArray is {0}", maxVal);
   Console.ReadKey();
}
```

он не будет компилироваться из-за того, что указан параметр не того типа. Позже в разделе "Перегрузка функций" будет описан один полезный прием для обхода этой проблемы.

Массивы параметров

Язык C# позволяет указывать один (и только один) специальный параметр для функции. Этот параметр называется *массивом параметров*; в определении функции он должен указываться последним. Массивы параметров позволяют вызывать функции с произвольным количеством параметров и определяются с помощью ключевого слова params.

Массивы параметров удобны для упрощения кода, поскольку при этом не нужно передавать массивы из вызывающего кода. Вместо этого передаются несколько однотипных параметров, к которым можно обращаться из функции.

Код для определения функции, использующей массив параметров, выглядит так:

```
static <возвращаемыйТип> <имяФункции>(<типПараметра1> <имяПараметра1>, ... ,
                                       params <тип>[] <имя>)
{
   ...
   return <возвращаемоеЗначение>;
}
```

Код вызова такой функции выглядит следующим образом:

```
<имяФункции>(<параметр1>, ..., <значение1>, <значение2>, ...)
```

Здесь <значение1>, <значение2> и т.д. представляют собой значения типа <тип>, которые используются для инициализации массива <имя>. Количество параметров, которые можно здесь указать, практически бесконечно; единственным ограничением является то, что все они обязательно должны иметь один и тот же <тип>. Можно даже вообще не указывать никаких параметров.

Последнее свойство делает массивы параметров особенно полезными для предоставления функциям дополнительной информации, которая может потребоваться во время обработки. Например, предположим, что есть функция по имени GetWord(), которая прини-

мает строку (значение string) в качестве первого параметра и возвращает первое слово из этой строки:

```
string firstWord = GetWord("This is a sentence.");
```

В данном случае переменной firstWord будет присвоена строка This.

Добавив в эту функцию параметр params, можно при желании возвращать какое-то другое слово, указав его индекс:

```
string firstWord = GetWord("This is a sentence.", 2);
```

Если отсчет слов начинается с 1 (для первого слова), в данном случае переменной firstWord будет присвоена строка is.

Помимо этого, с помощью params также можно ограничить количество возвращаемых символов, указывая его в третьем параметре:

```
string firstWord = GetWord("This is a sentence.", 4, 3);
```

В данном случае переменной firstWord будет присвоена строка sen.

В следующем практическом занятии демонстрируется определение и использование функции с параметром params.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Обмен данными с функцией (часть 2)

1. Создайте новое консольное приложение с именем Ch06Ex03 и сохраните его в каталоге C:\BegVCSharp\Chapter06.
2. Добавьте файл Program.cs следующий код:

```
class Program
{
   static int SumVals(params int[] vals)
   {
      int sum = 0;
      foreach (int val in vals)
      {
         sum += val;
      }
      return sum;
   }
   static void Main(string[] args)
   {
      int sum = SumVals(1, 5, 2, 9, 8);
      Console.WriteLine("Summed Values = {0}", sum); // Сумма значений
      Console.ReadKey();
   }
}
```

*Фрагмент кода **Ch06Ex03\Program.cs***

3. Запустите приложение. На рис. 6.3 показан результат, который должен получиться.

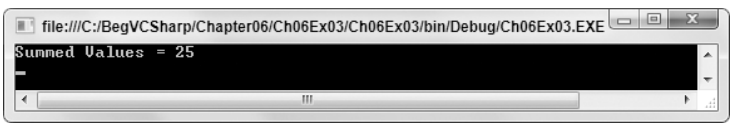


***Рис. 6.3.** Приложение Ch06Ex03 в действии*

Описание работы

В этом примере функция `SumVals()` определена с использованием ключевого слова `params`, чтобы она могла принимать любое количество параметров типа `int` (и никакие другие):

```
static int SumVals(params int[] vals)
{
  ...
}
```

Код в этой функции просто перебирает значения в массиве `vals` и суммирует их, а затем возвращает результат. В коде `Main()` осуществляется вызов этой функции с пятью целочисленными параметрами:

```
int sum = SumVals(1, 5, 2, 9, 8);
```

Обратите внимание, что эту функцию так же легко можно вызвать и без параметров или с одним, двумя, а то и с сотней целочисленных параметров — предела количеству передаваемых параметров не существует.

Параметры-ссылки и параметры-значения

Во всех определенных до сих пор функциях использовались *параметры-значения* (value parameters). То есть при каждом применении параметров используемой внутри функции переменной-параметру передавалось значение. Никакие изменения этой переменной в функции никак не влияют на параметр, указанный в вызове функции. Например, рассмотрим функцию, которая удваивает и выводит значение переданного ей параметра:

```
static void ShowDouble(int val)
{
  val *= 2;
  Console.WriteLine("Удвоенное значение val = {0}", val);
}
```

Эта функция удваивает значение параметра `val`. Если вызвать ее следующим образом:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

то выходные данные в окне консоли будут выглядеть так:

```
myNumber = 5
Удвоенное значение val = 10
myNumber = 5
```

Вызов `ShowDouble()` с `myNumber` в качестве параметра никак не влияет на значение `myNumber` в `Main()`, хотя параметр `val`, которому оно присваивается, умножается на 2.

Все это замечательно, но что делать, если все-таки *нужно*, чтобы значение `myNumber` изменялось? Кончено, можно использовать функцию, возвращающую новое значение для `myNumber`:

```
static int DoubleNum(int val)
{
  val *= 2;
  return val;
}
```

Для вызова этой функции можно использовать такой код:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = DoubleNum(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Однако такой код не очень понятен и не может изменить значения сразу нескольких переменных, передаваемых в качестве параметров (поскольку функции возвращают единственное значение).

Поэтому передавать параметр необходимо по *ссылке* (reference). Тогда функция будет работать именно с той переменной, которая указана в вызове функции, а не просто с переменной, которая имеет такое же значение. В результате любые изменения значения этой переменной будут отражаться и в переменной, переданной в качестве параметра. Для этого в описании параметра нужно указать ключевое слово `ref`:

```
static void ShowDouble(ref int val)
{
  val *= 2;
  Console.WriteLine("Удвоенное значение val = {0}", val);
}
```

Это же слово необходимо использовать в вызове функции (обязательно):

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Тогда текстовый вывод в окне консоли будет выглядеть так:

```
myNumber = 5
удвоенное значение val = 10
myNumber = 10
```

На этот раз функция `ShowDouble()` изменила значение `myNumber`.

Для переменных, передаваемых в качестве параметра `ref`, имеются два ограничения. Во-первых, функция может изменять значение параметра-ссылки, поэтому в вызове функции должны использоваться только *неконстантные* переменные. Следовательно, следующий код является недопустимым:

```
const int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Во-вторых, используемая переменная должна быть инициализирована на момент вызова функции. Язык C# не позволяет предполагать, что параметр `ref` будет инициализирован в использующей его функции. То есть следующий код тоже недопустим:

```
int myNumber;
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Выходные параметры

Помимо передачи значений по ссылке можно указывать, что данный параметр является *выходным*, используя ключевое слово `out`, которое применяется так же, как и ключевое слово `ref` — в виде модификатора перед параметром как в определении, так и в вызове функции. В сущности, он обеспечивает почти такое же поведение, как и параметр-ссылка, т.е. по окончании работы функции значение переменной возвращается в переменную, указанную при вызове функции. Однако существуют два важных отличия.

- в качестве параметра `ref` применять неинициализированную переменную нельзя, а в качестве параметра `out` — можно;
- в функции, которая использует параметр `out`, должно предполагаться, что ему не присвоено никакое значение.

Это означает, что хотя в вызывающем коде в качестве параметра `out` можно использовать инициализированную переменную, хранившееся в этой переменной значение пропадет при выполнении функции.

Для примера возьмем рассмотренную ранее функцию `MaxValue()`, которая возвращает максимальное значение в массиве. Изменим ее так, чтобы она позволяла получать индекс того элемента, в котором находится максимальное значение. Чтобы не усложнять пример, пусть при наличии нескольких элементов с таким же максимальным значением это будет индекс первого такого значения. Для этого достаточно добавить параметр `out`, изменив функцию следующим образом:

```
static int MaxValue(int[] intArray, out int maxIndex)
{
   int maxVal = intArray[0];
   maxIndex = 0;
   for (int i = 1; i < intArray.Length; i++)
   {
      if (intArray[i] > maxVal)
      {
         maxVal = intArray[i];
         maxIndex = i;
      }
   }
   return maxVal;
}
```

Использовать эту функцию можно так:

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxIndex;
Console.WriteLine("Максимальное значение в myArray равно {0}",
                  MaxValue(myArray, out maxIndex));

Console.WriteLine("Первый экземпляр этого значения находится в элементе {0}",
                  maxIndex + 1);
```

Это приведет к выводу следующего результата:

```
Максимальное значение в myArray равно 9
Первый экземпляр этого значения находится в элементе 7
```

Как и ключевое слово `ref`, ключевое слово `out` также обязательно указывать при вызове функции.

К возвращенному здесь значению `maxIndex` перед выводом была добавлена единица. Это сделано для преобразования индекса в более привычную форму, когда отсчет начинается с 1, а не с 0.

Область видимости переменных

При прочтении предыдущего раздела у вас, возможно, возник вопрос о том, зачем необходим обмен данными с функциями. Дело в том, что доступ к переменным в C# возможен только из локализованных областей кода. Говорят, что у каждой переменной имеется так называемая *область видимости* (scope), в пределах которой она доступна.

Область видимости переменных удобнее объяснить на примере. В следующем практическом занятии демонстрируется ситуация, когда выполняется определение переменной в одной области видимости и обращение к ней из другой области.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Область видимости переменной

1. Внесите в файл `Program.cs` приложения `Ch06Ex01` следующие изменения:

```
class Program
{
   static void Write()
   {
      Console.WriteLine("myString = {0}", myString);
   }
   static void Main(string[] args)
   {
      string myString = "String defined in Main()";
                        // Строка, определенная в Main()
      Write();
      Console.ReadKey();
   }
}
```

Фрагмент кода ***Ch06Ex01\Program.cs***

2. Скомпилируйте этот код и обратите внимание на сообщение об ошибке и предупреждение:

```
The name 'myString' does not exist in the current context
The variable 'myString' is assigned but its value is never used
Имя myString не существует в текущем контексте
Переменной myString присвоено значение, но оно не используется
```

Описание работы

Что здесь не так? Дело в том, что переменная `myString`, определенная в основном теле приложения (в функции `Main()`), недоступна из функции `Write()`.

Объясняется это тем, что у каждой переменной имеется область видимости, в пределах которой она действительна. Область видимости охватывает блок кода, в котором переменная определена, а также все непосредственно вложенные в него блоки. Блоки кода в функциях отделены от блоков кода, из которых они вызываются. Внутри функции `Write()` имя `myString` нигде не определено, а это значит, что переменная `myString`, определенная в `Main()`, находится *за пределами области видимости* — она может использоваться только в пределах функции `Main()`. Однако в функции `Write()` может иметься своя, совершенно отдельная переменная с именем `myString`. Попробуйте изменить предыдущий код следующим образом:

```
class Program
{
   static void Write()
   {
      string myString = "String defined in Write()";
                       // Строка, определенная в функции Write()
      Console.WriteLine("Now in Write()");
                        // Теперь в функции Write()
      Console.WriteLine("myString = {0}", myString);
   }
   static void Main(string[] args)
   {
      string myString = "String defined in Main()";
      Write();
      Console.WriteLine("\nNow in Main()");
      Console.WriteLine("myString = {0}", myString);
      Console.ReadKey();
   }
}
```

Этот код успешно скомпилируется и выдаст результаты, показанные на рис. 6.4.

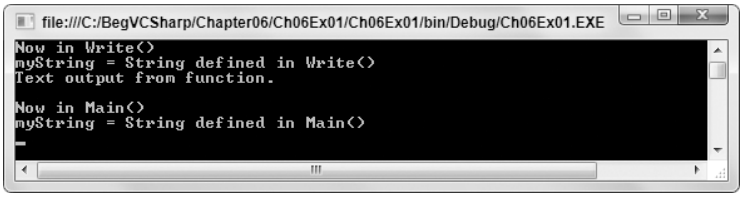


Рис. 6.4. *Результаты после изменения кода*

Ниже перечислены действия, выполняемые этим кодом.

- Функция `Main()` определяет и инициализирует строковую переменную с именем `myString`.
- Функция `Main()` передает управление функции `Write()`.
- Функция `Write()` определяет и инициализирует строковую переменную `myString`, отличную от `myString`, которая определена в `Main()`.
- Функция `Write()` выводит на консоль строку, содержащую значение той `myString`, которая была определена в `Write()`.
- Функция `Write()` возвращает управление функции `Main()`.
- Функция `Main()` выводит на консоль строку, содержащую значение той переменной `myString`, которая была определена в `Main()`.

Переменные, область видимости которых распространяется подобным образом только на единственную функцию, называются *локальными переменными*. Однако допустимо использовать и *глобальные переменные*, область видимости которых распространяется на несколько функций. Попробуйте изменить код следующим образом:

```
class Program
{
  static string myString;

  static void Write()
  {
    string myString = "String defined in Write()";
    Console.WriteLine("Now in Write()");
    Console.WriteLine("Local myString = {0}", myString);          // локальная
    Console.WriteLine("Global myString = {0}", Program.myString); // глобальная
  }

  static void Main(string[] args)
  {
    string myString = "String defined in Main()";
    Program.myString = "Global string";
    Write();
    Console.WriteLine("\nNow in Main()");
    Console.WriteLine("Local myString = {0}", myString);          // локальная
    Console.WriteLine("Global myString = {0}", Program.myString); // глобальная
    Console.ReadKey();
  }
}
```

Результат, который получится в этом случае, показан на рис. 6.5.

Здесь была добавлена еще одна переменная с именем `myString`, на этот раз на уровень выше в иерархии имен в коде.

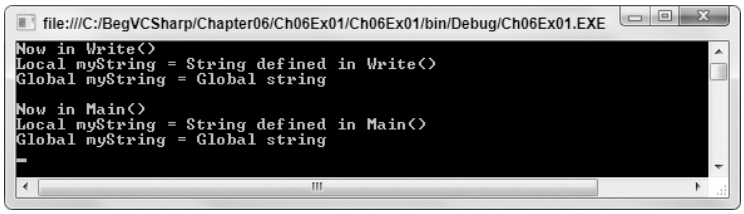


Рис. 6.5. *Использование локальных и глобальных переменных*

Определение этой переменной выглядит следующим образом:

```
static string myString;
```

Здесь также необходимо ключевое слово `static`. Вкратце это объясняется тем, что в таких консольных приложениях для подобных глобальных переменных обязательно нужно либо ключевое слово `static`, либо ключевое слово `const`. Если значение глобальной переменной должно изменяться, следует применять ключевое слово `static`, поскольку `const` не допускает изменения значения переменной.

Чтобы отличить такую глобальную переменную от одноименных локальных переменных в функциях `Main()` и `Write()`, необходимо использовать для нее полностью определенное имя, как было описано в главе 3. Здесь для обращения к глобальной переменной используется имя `Program.myString`. Подобное имя необходимо только в том случае, когда есть и глобальные, и локальные переменные с одинаковыми именами; если бы локальной переменной `myString` не было, для обращения к глобальной переменной можно было бы писать просто `myString`, а не `Program.myString`. При наличии локальной переменной с таким же именем, как у глобальной, глобальная переменная называется *скрытой*.

Значение глобальной переменной устанавливается в функции `Main()` с помощью строки

```
Program.myString = "Global string";
```

а обращение в функции `Write()` — следующим образом:

```
Console.WriteLine("Global myString = {0}", Program.myString);
```

Может возникнуть вопрос: почему нельзя просто использовать этот прием для обмена данными с функциями вместо передачи параметров? Вообще-то бывают ситуации, когда такой подход действительно более удобен, но чаще это не так. Выбор, применять ли глобальные переменные, зависит от предполагаемого использования конкретной функции. Глобальные переменные неудобны тем, что они обычно не пригодны для "универсальных" функций, которые могут работать с любыми передаваемыми им данными, а не только с содержимым конкретной глобальной переменной. Об этом более подробно будет рассказано несколько позже.

Область видимости переменных в других структурах

Один из описанных в предыдущем разделе моментов относится не только к области видимости переменных между функциями: области видимости переменных охватывают блоки кода, в которых они определены, и все непосредственно вложенные в них блоки. Это касается и других блоков, вроде тех, что содержатся в структурах ветвлений и циклов. Рассмотрим следующий код:

```
int i;
for (i = 0; i < 10; i++)
{
  string text = "Line " + Convert.ToString(i);
  Console.WriteLine("{0}", text);
}
```

```
Console.WriteLine("Last text output in loop: {0}", text);
                  // Последний текст, выведенный в цикле
```

Здесь строковая переменная `text` локальна в цикле `for`. Скомпилировать этот код не получится, поскольку вызов функции `Console.WriteLine()` за пределами данного цикла приведет к попытке использования переменной `text` за пределами ее области видимости. Попробуем изменить этот код следующим образом:

```
int i;
string text;
for (i = 0; i < 10; i++)
{
  text = "Line " + Convert.ToString(i);
  Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

Этот код также содержит ошибку: переменные перед использованием должны обязательно объявляться и инициализироваться, а переменная `text` инициализируется только в цикле `for`. Поэтому после выхода из цикла присвоенное ей значение теряется. Однако можно внести в код такое изменение:

```
int i;
string text = "";
for (i = 0; i < 10; i++)
{
  text = "Line " + Convert.ToString(i);
  Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

На этот раз переменная `text` инициализирована за пределами цикла, и доступ к ее значению возможен. Результат выполнения этого простого кода показан на рис. 6.6.

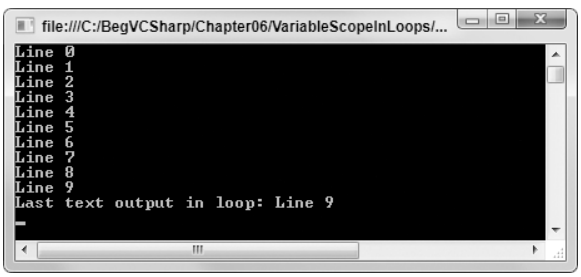


Рис. 6.6. *Результат выполнения измененного кода*

Здесь последнее значение, присвоенное `text` в цикле, доступно и за пределами цикла. Понимание данной темы требует несколько больших усилий. На первый взгляд непонятно, почему, с учетом предыдущего примера, у переменной `text` после завершения цикла не остается в качестве значения пустая строка, которая была присвоена ей перед входом в цикл.

Причина такого поведения связана с выделением памяти для переменной `text` — как и для любой другой переменной. Одно лишь объявление переменной простого типа ни к каким действиям не приводит. Место в памяти выделяется только при присваивании переменным сохраняемых значений. Если это выделение памяти происходит внутри цикла, значение, по сути, определяется как локальное значение, и вне цикла выходит за границы своей области видимости.

Даже если сама переменная не локализована в цикле, это не относится к содержащемуся в ней значению. Однако присваивание значения переменной за пределами цикла делает его локальным по отношению к главному коду и видимым внутри цикла. Это означает, что переменная не выходит за границы области видимости до завершения основного блока кода, поэтому ее значение доступно и за пределами цикла.

К счастью для программистов, компилятор C# обнаруживает проблемы, связанные с областью видимости переменных, а внимательный анализ генерируемых им сообщений об ошибках позволяет лучше разобраться в этой теме.

И, наконец, рассмотрим рекомендуемый подход. Как правило, лучше объявлять и инициализировать все переменные перед теми блоками кода, в которых они будут использоваться. Исключением являются лишь переменные-счетчики цикла — их часто удобнее объявлять внутри представляющего цикл блока:

```
for (int i = 0; i < 10; i++)
{
   ...
}
```

Здесь `i` локализуется в блоке цикла, но в это нет ничего страшного, поскольку доступ к таким переменным-счетчикам из внешнего кода требуется крайне редко.

Сравнение параметров и возвращаемых значений с глобальными данными

Рассмотрим более подробно преимущества и недостатки обмена данными с функциями через глобальные данные и через параметры и возвращаемые значения. Взгляните на следующий код:

```
class Program
{
   static void ShowDouble(ref int val)
   {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val); // удвоенное значение val
   }

   static void Main(string[] args)
   {
      int val = 5;
      Console.WriteLine("val = {0}", val);
      ShowDouble(ref val);
      Console.WriteLine("val = {0}", val);
   }
}
```

Этот код немного отличается от приведенного ранее примера использования переменной `myNumber` в функции `Main()`. Он демонстрирует, что локальные переменные могут иметь одинаковые имена и при этом совершенно не мешать друг другу. Это также означает, что два приведенных здесь примера кода имеют больше сходства, чем различий, а это позволяет сконцентрироваться на конкретных отличиях и не обращать внимания на имена переменных.

Теперь сравните его с таким кодом:

```
class Program
{
   static int val;
```

```
    static void ShowDouble()
    {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val); // удвоенное значение val
    }
    static void Main(string[] args)
    {
      val = 5;
      Console.WriteLine("val = {0}", val);
      ShowDouble();
      Console.WriteLine("val = {0}", val);
    }
}
```

Результаты вызова обеих приведенных здесь функций `ShowDouble` совпадают.

Никаких жестких правил, когда следует применять тот, а когда другой подход, не существует. Оба они вполне допустимы, хотя существуют следующие рекомендации.

Прежде всего, как уже упоминалось при первоначальном обсуждении данной темы, версия `ShowDouble()`, в которой используется глобальное значение, требует обязательного применения глобальной переменной `val`. Это ограничивает универсальность функции и означает, что для сохранения результатов придется постоянно копировать значение этой глобальной переменной в другие переменные. Кроме того, глобальные данные могут быть изменены и в каком-нибудь месте приложения, что может привести к непредсказуемым результатам (например, незаметному изменению значений до тех пор, пока не станет слишком поздно).

Однако такая потеря универсальности зачастую может оказаться преимуществом. Нередко функция используется только для какой-то одной цели, а применение хранилища глобальных данных снижает вероятность появления ошибки в вызове функции, например, передачи не той переменной.

Конечно, можно возразить, что подобная простота на самом деле снижает наглядность кода. Явное указание параметров позволяет сразу видеть, что именно изменяется в функции. При виде вызова наподобие `myFunction(val1, out val2)` сразу понятно, что `val1` и `val2` — важные переменные, на которые следует обратить внимание, и что по завершении выполнения функции переменной `val2` будет присвоено новое значение. Но если функция не имеет параметров, то понять, какие данные обрабатываются внутри и как, невозможно.

И еще — учтите, что применение глобальных данных возможно не всегда. Далее в этой книге имеются примеры кода, хранящегося в разных файлах и/или относящегося к разным пространствам имен, которые взаимодействуют друг с другом посредством функций. В подобных случаях код часто разбивается до такой степени, что очевидного варианта для размещения глобального хранилища просто нет.

Для обмена данными можно свободно применять любой из описанных подходов. Чаще лучше использовать параметры, а не глобальные данные, но бывают очевидные случаи, когда удобнее глобальные данные, и их использование точно не будет ошибкой.

Функция `Main()`

Теперь, когда уже рассмотрено большинство простых приемов, применяемых для создания и использования функций, можно более подробно познакомиться с функцией `Main()`.

Ранее уже было сказано, что функция `Main()` является точкой входа в любом C#-приложении, и что ее выполнение охватывает выполнение всего приложения. То есть запуск приложения приводит к началу выполнения функции `Main()`, а при завершении работы `Main()` прекращается выполнение приложения.

Функция `Main()` может возвращать либо `void`, либо тип `int`, а также принимать необязательный параметр `string[] args`, т.е. ее можно использовать ее в любом из следующих вариантов:

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

В третьем и четвертом варианте она возвращает значение `int`, которое позволяет сообщить, как завершена работа приложения, и зачастую применяется как признак ошибки (хотя это совсем не обязательно). Как правило, возврат значения 0 означает нормальное завершение (т.е. приложение закончило работу и может быть безопасно завершено).

Необязательный параметр `args` в функции `Main()` позволяет передавать в приложение информацию извне. Эта информация задается в виде *параметров командной строки*.

Вы уже наверняка встречались с параметрами командной строки. При запуске приложения из командной строки ему часто можно непосредственно указать некоторую информацию – например, какой файл требуется загрузить для обработки приложением. Возьмем известное Windows-приложение Notepad (Блокнот). Это приложение можно запустить, введя команду **`Notepad`** в окне командной строки или в окне, которое появляется при выборе в меню Start (Пуск) пункта Run (Выполнить). Вместо **`Notepad`** в таком окне можно ввести и что-то вроде **`Notepad "myfile.txt"`**. Тогда приложение Notepad при запуске либо сразу же загрузит файл с именем `myfile.txt`, либо предложит создать файл с таким именем, если он не существует. В данном случае `"myfile.txt"` является аргументом командной строки. Параметр `args` как раз и позволяет обеспечивать такое поведение консольных приложений.

При выполнении консольного приложения все параметры, указанные в командной строке, помещаются в массив `args`. После этого их можно использовать в приложении. В следующем практическом занятии демонстрируются соответствующий пример. Аргументы командной строки можно задавать в любом количестве, и все они будут выведены в окне консоли.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Аргументы командной строки

1. Создайте новое консольное приложение с именем `Ch06Ex04` и сохраните его в каталоге `C:\BegVCSharp\Chapter06`.

2. Добавьте в файл `Program.cs` следующий код:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("{0} command line arguments were specified:", args.Length);
                  // Задано {0} аргументов командной строки
    foreach (string arg in args)
       Console.WriteLine(arg);
    Console.ReadKey();
  }
}
```

*Фрагмент кода **`Ch06Ex04\Program.cs`***

3. Откройте окно со страницами свойств проекта (щелкнув правой кнопкой мыши на имени проекта `Ch06Ex04` в окне Solution Explorer (Проводник решений) и выбрав в контекстном меню пункт Properties (Свойства)).

4. Перейдите на страницу Debug (Отладка) и добавьте в поле Command Line Arguments (Аргументы командной строки) любые аргументы — например, так, как показано на рис. 6.7.

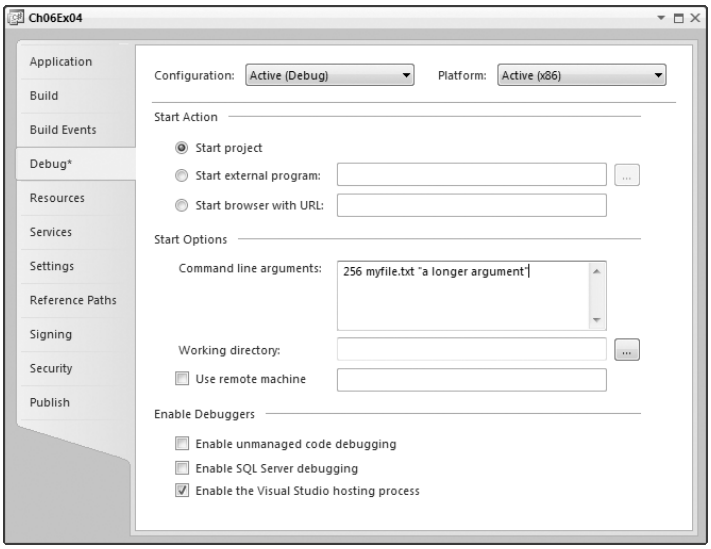


Рис. 6.7. *Добавление аргументов командной строки*

5. Запустите приложение. На рис. 6.8 показан результат, который должен получиться.

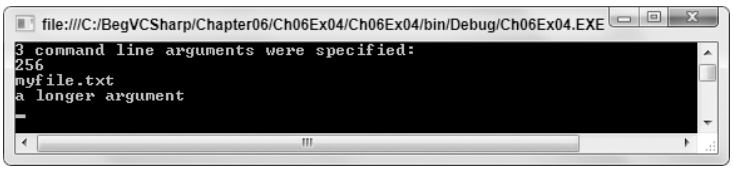


Рис. 6.8. *Приложение* `Ch06Ex04` *в действии*

Описание работы

Приведенный здесь код очень прост:

```
Console.WriteLine("{0} command line arguments were specified:", args.Length);
foreach (string arg in args)
  Console.WriteLine(arg);
```

Параметр `args` просто используется так же, как и любой другой строковый массив. Никаких сложных действий над аргументами не выполняется; все они просто выводятся на экран. В данном случае эти аргументы переданы с помощью свойств проекта в IDE-среде. Это удобный способ использовать одни и те же аргументы командной строки при каждом запуске приложения из IDE-среды вместо того, чтобы каждый раз заново вводить их в командной строке. Точно такого же результата можно добиться и открыв окно командной строки в том же каталоге, что и выходные данные проекта (а именно — `C:\BegCSharp\Chapter6\Ch06Ex04\Ch06Ex04\bin\Debug`), и введя следующую команду:

```
Ch06Ex04 256 myFile.txt "a longer argument"
```

Все аргументы отделяются друг от друга пробелами. Если нужно передать аргумент, содержащий пробелы, то его следует заключить в двойные кавычки, и тогда он не будет воспринят как несколько аргументов.

Функции в структурах

В предыдущей главе были описаны типы структур, которые позволяют хранить несколько элементов данных в одном месте. На самом деле структуры способны на гораздо большее. Например, они могут содержать не только данные, но и функции. Это свойство может показаться странным, но в действительности оно весьма полезно. В качестве простого примера рассмотрим следующую структуру:

```
struct customerName
{
   public string firstName, lastName;
}
```

Если нужно вывести в окно консоли полное имя, то это имя придется собирать из отдельных частей. Для переменной типа `CustomerName` с именем `myCustomer` можно воспользоваться, например, таким синтаксисом:

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine("{0} {1}", myCustomer.firstName, myCustomer.lastName);
```

Добавив в структуру соответствующие функции, можно упростить и централизовать решение общих задач — например, следующим образом:

```
struct customerName
{
   public string firstName, lastName;

   public string Name ()
   {
      return firstName + " " + lastName;
   }

}
```

Эта функция выглядит примерно так же, как и любая другая функция из этой главы; единственное отличие состоит в том, что в ней не применяется модификатор `static`. Причины будут объяснены позже, а пока достаточно знать, что для функций в структурах данное ключевое слово не требуется. Применить эту функцию можно следующим образом:

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine(myCustomer.Name());
```

Такой синтаксис и гораздо проще, и гораздо понятнее предыдущего. Функция `Name()` имеет прямой доступ к членам структуры `firstName` и `lastName`. В пределах структуры `customerName` они могут считаться глобальными.

Перегрузка функций

Ранее в этой главе уже говорилось о необходимости соответствия с сигнатурой функции при ее вызове. Это означает, что для выполнения операций над разными типами переменных нужны отдельные функции. Механизм *перегрузки функций* (function overloading)

позволяет создавать несколько функций, имеющих одинаковое имя, но работающих с разными типами параметров. Например, ранее приводился код с функцией `MaxValue()`:

```
class Program
{
  static int MaxValue(int[] intArray)
  {
    int maxVal = intArray[0];
    for (int i = 1; i < intArray.Length; i++)
    {
      if (intArray[i] > maxVal)
      maxVal = intArray[i];
    }
    return maxVal;
  }
  static void Main(string[] args)
  {
    int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
    int maxVal = MaxValue(myArray);
    Console.WriteLine("The maximum value in myArray is {0}", maxVal);
                    // Максимальное значение myArray равно
    Console.ReadKey();
  }
}
```

Эту функцию можно использовать только с массивами значений `int`. Для работы с другими типами параметров можно было бы дописать соответствующие функции с другими именами (т.е., например, изменить имя этой функции на `IntArrayMaxValue()` и добавить функции с именами вроде `DoubleArrayMaxValue()` для обработки других типов), либо добавить в код следующую функцию:

```
...
static double MaxValue(double[] doubleArray)
{
  double maxVal = doubleArray[0];
  for (int i = 1; i < doubleArray.Length; i++)
  {
    if (doubleArray[i] > maxVal)
    maxVal = doubleArray[i];
  }
  return maxVal;
}
...
```

Единственным отличием здесь является применение значений `double`. Имя функции — `MaxValue()` — в точности то же, но вот ее *сигнатура* (что очень важно) выглядит по-другому. Ведь, как уже было сказано, в сигнатуру функции входит как имя функции, так и ее параметры. Определение двух функций с одинаковыми именами и сигнатурами было бы ошибкой, но эти две функции имеют разные сигнатуры, и ошибки здесь нет.

Возвращаемый тип функции не является частью ее сигнатуры, поэтому определять две функции, отличающиеся только типом возвращаемого значения, нельзя; их сигнатуры считаются идентичными.

После добавления приведенного выше кода функция `MaxValue()` будет представлена в двух версиях: одна принимает в качестве входных данных массив значений типа `int` и возвращает максимальное значение в нем как число типа `int`, а вторая принимает в качестве входных данных массив значений типа `double` и возвращает максимальное значения в нем в виде числа типа `double`.

Замечательно то, что не требуется явно указывать, какая именно из этих двух версий должна использоваться. Достаточно просто передать параметр типа массива, и нужная версия этой функции будет выбрана автоматически в зависимости от типа параметра.

Обратите внимание на еще одно свойство механизма IntelliSense в VS и VCE: при наличии в приложении двух таких версий функции и вводе ее имени IDE будет отображать все доступные перегрузки для данной функции. Например, после ввода строки

```
double result = MaxValue(
```

IDE-среда отобразит информацию об обеих версиях функции `MaxValue()`, между которыми можно переключаться с помощью клавиш со стрелками вверх и вниз (рис. 6.9).

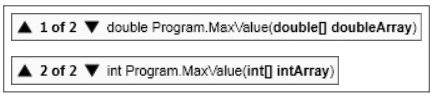


Рис. 6.9. *Информация о доступных перегруженных версиях функции*

При перегрузке функций учитываются все аспекты их сигнатур. Например, может существовать две разных версии функции, принимающие параметры по значению и по ссылке:

```
static void ShowDouble(ref int val)
{
   ...
}
static void ShowDouble(int val)
{
   ...
}
```

Определение, какая версия должна использоваться, выполняется на основании того, содержится ли в вызове функции ключевое слово `ref` или нет. Следующая строка приведет к вызову той версии, которая принимает параметры по ссылке:

```
ShowDouble(ref val);
```

а вот эта — версию с передачей параметра по значению:

```
ShowDouble(val);
```

Можно сделать и так, чтобы версии функции отличались друг от друга количеством параметров и т.д.

Делегаты

Делегат (delegate) — это тип, который позволяет хранить ссылки на функции. Возможно, это звучит довольно замысловато, но на самом деле механизм делегатов на удивление прост. Основное назначение делегатов будет объяснено позже при рассмотрении событий и их обработки, но полезно рассказать здесь о них хотя бы вкратце. Объявляются делегаты во многом так же, как функции, но без тела функции и с ключевым словом `delegate`. В объявлении делегата указывается возвращаемый тип и список параметров.

После определения делегата можно объявлять переменную с типом этого делегата. Затем эту переменную можно инициализировать как ссылку на любую функцию, которая имеет такой же возвращаемый тип и список параметров, как и у делегата. После этого функцию можно вызывать с помощью переменной-делегата, как будто это и есть сама функция.

Наличие переменной со ссылкой на функцию позволяет выполнять и другие операции, которые иначе были бы невозможны. Например, можно передать переменную-делегат в качестве параметра функции, и тогда во время выполнения можно вызывать функцию с помощью этого делегата. В следующем практическом занятии демонстрируется применение делегата для обращения к одной из двух возможных функций.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Вызов функции с помощью делегата

1. Создайте новое консольное приложение с именем `Ch06Ex05` и сохраните его в каталоге `C:\BegVCSharp\Chapter06`.

2. Добавьте в файл `Program.cs` следующий код:

```
class Program
{
   delegate double ProcessDelegate(double param1, double param2);

   static double Multiply(double param1, double param2)
   {
      return param1 * param2;
   }
   static double Divide(double param1, double param2)
   {
      return param1 / param2;
   }
   static void Main(string[] args)
   {
      ProcessDelegate process;
      Console.WriteLine("Enter 2 numbers separated with a comma:");
                        // Введите 2 числа через запятую
      string input = Console.ReadLine();
      int commaPos = input.IndexOf(',');
      double param1 = Convert.ToDouble(input.Substring(0, commaPos));
      double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                                 input.Length - commaPos - 1));
      Console.WriteLine("Enter M to multiply or D to divide:");
                        // Введите M для умножения или D для деления
      input = Console.ReadLine();
      if (input == "M")
         process = new ProcessDelegate(Multiply);
      else
         process = new ProcessDelegate(Divide);
      Console.WriteLine("Result: {0}", process(param1, param2));
                        // Результат
      Console.ReadKey();
   }
}
```

Фрагмент кода ***Ch06Ex05\Program.cs***

3. Запустите приложение. На рис. 6.10 показан результат, который должен получиться.

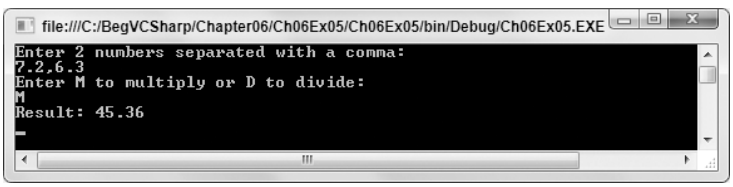


Рис. 6.10. *Приложение* `Ch06Ex05` *в действии*

Описание работы

В приведенном коде определяется делегат с именем `ProcessDelegate`, возвращаемый тип и параметры которого соответствуют двум функциям: `Multiply()` и `Divide()`.

Вот определение этого делегата:

```
delegate double ProcessDelegate(double param1, double param2);
```

Ключевое слово `delegate` указывает, что определяется делегат, а не функция (ведь это определение расположено в том месте, где могло бы находиться определение функции). Далее в определении указывается возвращаемый тип `double` и два параметра `double`. Использованные имена выбраны произвольно; типам и параметрам делегатов можно давать любые имена. В данном коде для делегата было выбрано имя `ProcessDelegate`, а параметры типа `double` названы `param1` и `param2`.

Код в функции `Main()` начинается с объявления переменной нового типа делегата:

```
static void Main(string[] args)
{
  ProcessDelegate process;
```

Далее идет довольно стандартный код на C#, который сначала запрашивает два разделенных запятой числа и затем помещает их в две переменные `double`:

```
Console.WriteLine("Enter 2 numbers separated with a comma:");
string input = Console.ReadLine();
int commaPos = input.IndexOf(',');
double param1 = Convert.ToDouble(input.Substring(0, commaPos));
double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                          input.Length - commaPos - 1));
```

Чтобы не загромождать код, в него не включена операция проверки правильности вводимых пользователем данных. В реальных ситуациях такую проверку нужно делать обязательно, хотя это может сильно увеличить объем кода.

Далее пользователю предлагается перемножить или поделить эти числа:

```
Console.WriteLine("Enter M to multiply or D to divide:");
input = Console.ReadLine();
```

После этого, в зависимости от выбора, инициализируется переменная делегата `process`:

```
if (input == "M")
  process = new ProcessDelegate(Multiply);
else
  process = new ProcessDelegate(Divide);
```

Для присваивания переменной-делегату ссылки на функцию применяется несколько необычный синтаксис. Примерно так же, как и при присваивании значений массивам, при создании нового делегата должно использоваться ключевое слово `new`. После `new` указывается тип делегата и параметр, указывающий на нужную функцию — в данном случае `Multiply()` или `Divide()`. Этот параметр не соответствует ни параметрам типа делегата, ни параметрам целевой функции; такой синтаксис применяется исключительно для делегатов. На его месте указывается просто имя применяемой функции, безо всяких круглых скобок.

Вообще-то при желании можно использовать и несколько более простой синтаксис:

```
if (input=="M")
  process=Multiply;
else
  process=Divide;
```

Компилятор определяет, что тип делегата переменной-процесса соответствует сигнатуре двух функций и автоматически инициализирует делегат. Вы можете выбрать любой вариант синтаксиса, хотя некоторые считают, что более длинный вариант понятнее.

И в конце осуществляется вызов выбранной функции с помощью делегата. Здесь применяется один и тот же синтаксис, независимо от того, на какую из функций ссылается делегат:

```
  Console.WriteLine("Result: {0}", process(param1, param2));
  Console.ReadKey();
}
```

Переменная-делегат обрабатывается так, как если бы это было имя функции. Но в отличие от функции, с ней можно также выполнять и другие действия — например, передавать ее функции через параметр, как показано в следующем примере:

```
static void ExecuteFunction(ProcessDelegate process)
{
  process(2.2, 3.3);
}
```

Это означает, что управлять поведением функций можно с помощью передачи им делегатов — подобно выбору наиболее подходящей для использования "оснастки". Например, может существовать функция, сортирующая массивы строк в алфавитном порядке. Для сортировки списков могут применяться разнообразные алгоритмы с разной производительностью, что зависит от характеристик сортируемого списка. С помощью делегатов можно выбирать используемый алгоритм, передавая функции сортировки делегат алгоритма сортировки.

У делегатов имеется еще много подобных сфер применения, но, как было сказано, наиболее важной из них является обработка событий, которая рассматривается в главе 13.

Резюме

В этой главе был предоставлен довольно подробный обзор способов применения функций в программах на C#. Многие из дополнительных возможностей, которые предлагают функции (особенно делегаты), более абстрактны, но их понимание необходимо для освоения объектно-ориентированного программирования — об этом речь пойдет в главе 8.

Умение использовать функции играет центральную роль во всех видах программирования. В последующих главах, особенно с главы 8, где содержится введение в объектно-ориентированное программирование, вы познакомитесь с более формальной структурой функций и их взаимосвязью с классами. Тогда станет ясно, что возможность абстрагирования кода в пригодные для многократного использования блоки является одной из самых полезных возможностей программирования на языке C#.

Упражнения

1. Две приведенных ниже функции содержат ошибки. Назовите их.

```
static bool Write()
{
  Console.WriteLine("Text output from function.");
}
static void myFunction(string label, params int[] args, bool showLabel)
{
  if (showLabel)
    Console.WriteLine(label);
  foreach (int i in args)
    Console.WriteLine("{0}", i);
}
```

2. Напишите приложение, которое использует два аргумента командной строки для занесения значений, соответственно, в строку и в целочисленную переменную, а затем выводит эти значения.

3. Создайте делегат и используйте его для выполнения функции `Console.ReadLine()` при запросе у пользователя входных данных.
4. Измените следующую структуру, чтобы она включала функцию, возвращающую общую цену заказа:

```
struct order
{
  public string itemName;       // Название товара
  public int    unitCount;      // Количество единиц
  public double unitCost;       // Цена за единицу
}
```

5. Добавьте в структуру `order` еще один член — функцию, возвращающую форматированную строку (с заменой всех курсивных элементов в угловых скобках соответствующими значениями):

```
Order Information: <количество единиц> <название товара> items
  at $ <цена за единицу> each,
  total cost $ <общая стоимость>
Информация о заказе: <количество единиц> штук <название товара>
  по цене <цена за единицу> каждый,
  общей стоимостью <общая стоимость>
```

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Определение функций	В определении функции указываются ее имя, ноль или более параметров и возвращаемый тип. Имя и параметры функции вместе определяют ее сигнатуру. Можно определять несколько функций с различными сигнатурами, но с одинаковыми именам — это называется перегрузкой функций. Кроме того, функции можно определять внутри структур.
Возвращаемые значения и параметры	Тип, возвращаемый функцией, может быть любым типом или `void`, если функция не возвращает значение. Параметры также могут быть любого типа, они определяются с помощью списка пар типа и имени, разделяемых запятыми. При вызове функции все указанные параметры должны соответствовать параметрам, указанным в определении — как по типу, так и по порядку в списке. Массив параметров позволяет указывать произвольное количество параметров указанного типа. Параметры можно описать как `ref` или `out`, это позволяет возвращать значения вызывающему блоку.
Области видимости переменных	Переменные локализованы в блоках кода, в которых они определены. К блокам кода относятся методы и другие структуры наподобие тела цикла. Можно определять несколько отдельных переменных с одинаковыми именами, но различными уровнями видимости.
Параметры командной строки	Функция `Main()` в консольном приложении может принимать параметры, которые передаются приложению из командной строки перед выполнением. Эти параметры разделяются запятыми, а длинные параметры можно заключать в кавычки.
Делегаты	Кроме непосредственного вызова функций, их можно вызывать с помощью делегатов. Делегаты — это переменные, которые определяются с возвращаемым типом и списком параметров. Конкретному типу делегата может соответствовать любой метод с таким же возвращаемым типом и параметрами, как и в определении этого делегата.

7

Отладка и обработка ошибок

В ЭТОЙ ГЛАВЕ...

- Методы отладки, доступные в IDE-среде
- Способы обработки ошибок, доступные в языке C#

До сих пор в книге рассматривались основы простого программирования на C#. Прежде чем перейти к объектно-ориентированному программированию в следующей части книги, необходимо ознакомиться с приемами отладки и обработки ошибок в коде C#.

Ошибки в коде неизбежны. Каким бы хорошим ни был программист, проблемы всегда возникают, и хороший программист должен осознавать это и быть готовым к их устранению. Конечно, некоторые проблемы являются незначительными и никак не влияют на работу приложения, как, например, ошибка в надписи на кнопке, но возможны и серьезные ошибки, приводящие к полной неработоспособности приложений (из-за чего они обычно называются *фатальными ошибками*). К числу фатальных ошибок относятся как простые ошибки в коде, которые препятствуют компиляции (*синтаксические* ошибки), так и более серьезные ошибки, которые проявляются только во время выполнения. Ошибки бывают весьма неочевидными. Например, приложение может не добавить запись в базу данных из-за отсутствия нужного поля или добавить запись с неверными данными по какой-то другой причине. Такие ошибки, при которых каким-либо образом нарушается логика приложения, называются *семантическими* или *логическими* ошибками.

Зачастую программист узнает о неочевидных ошибках лишь тогда, когда пользователь заявляет, что в приложении что-то работает не так. Тогда приходится анализировать код, выяснять, в чем дело, и устранять проблему, чтобы все работало как надо. В таких случаях имеющиеся в VS и VCE средства отладки оказывают огромную помощь. В первых разделах настоящей главы будут описаны некоторые из этих средств и способы их применения в типичных ситуациях.

Затем будут рассмотрены средства обработки ошибок, доступные в C#. Они позволяют принимать меры в тех случаях, когда вероятно возникновение ошибок, и писать код, достаточно устойчивый к ошибкам, которые в противном случае могли бы стать фатальными. Эти средства являются частью языка C#, а не механизма отладки, но в IDE предусмотрены инструменты для оказания помощи программисту и в этом.

Отладка в VS и VCE

Вы уже знаете, что приложения можно запускать двумя способами: с включенным и выключенным режимом отладки. По умолчанию при запуске приложения из VS или VCE оно запускается с включенным режимом отладки. Это происходит, например, при нажатии клавиши <F5> или щелчке на кнопке Play (Воспроизвести) с зеленой стрелкой в панели инструментов. Запустить приложение с выключенным режимом отладки можно, выбрав в меню Debug (Отладка) пункт Start Without Debugging (Запуск без отладки).

VS и VCE позволяют создавать приложения в двух конфигурациях: *отладочной* (Debug), используемой по умолчанию, и *рабочей* (Release). (Вообще-то можно определять и дополнительные конфигурации, но мы не будем углубляться в такие тонкости.) Переключаться между этими двумя конфигурациями можно с помощью раскрывающегося списка Solution Configurations (Конфигурации решения) в стандартной панели инструментов.

В VCE этот раскрывающийся список по умолчанию отключен. Для проработки материала настоящей главы его необходимо активизировать. Выберите в меню Tools (Сервис) пункт Options (Параметры), в открывшемся диалоговом окне Options (Параметры) установите флажок Show All Settings (Показывать все параметры), перейдите в категорию Projects and Solutions (Проекты и решения), и в подкатегории General (Общие) установите флажок Show Advanced Build Configurations (Показывать расширенные конфигурации сборки).

При сборке приложения в отладочной конфигурации и его запуске в отладочном режиме происходит не просто выполнение кода. В отладочных сборках сохраняется *информа-*

ция о символах приложения, чтобы IDE-среда точно знала, что происходит при выполнении каждой строки кода. Информация о символах нужна, например, для отслеживания имен переменных, которые используются в некомпилированном коде, чтобы их можно было связать со значениями в скомпилированном коде, где нет понятной человеку информации. Эта информация хранится в `.pdb`-файлах, которые вы, возможно, уже видели в каталогах `Debug`, и позволяет выполнять множество полезных действий.

- Выводить отладочную информацию в IDE-среде.
- Просматривать (и изменять) значения переменных в области видимости во время выполнения приложения.
- Приостанавливать и возобновлять выполнение программы.
- Автоматически останавливать выполнение программы в определенных точках кода.
- Выполнять программу построчно.
- Следить за изменениями в содержимом переменных во время выполнения приложения.
- Изменять содержимое переменных во время выполнения.
- Выполнять тестовые вызовы функций.

В рабочей конфигурации код приложения оптимизирован, и выполнять все эти действия невозможно. Однако рабочие сборки работают быстрее, и по окончании разработки приложения пользователям, как правило, передаются именно они, потому что они не требуют включения информации о символах, которая помещается в отладочные сборки.

В этом разделе описаны приемы отладки, позволяющие выявлять и исправлять области кода, которые работают не так, как ожидается — это и называется процессом *отладки*. В соответствии со способами их применения эти приемы сгруппированы в два раздела. В целом процесс отладки производится либо с помощью прерываний выполнения программы, либо с помощью формирования протокола работы для последующего анализа. В терминологии VS и VCE, это означает, что приложение либо работает, либо находится в режиме останова, т.е. его обычное выполнение приостановлено. Сначала мы рассмотрим приемы отладки в непрерывном (обычном) режиме.

Отладка в непрерывном (обычном) режиме

Одной из функций, которая использовалась повсюду в книге, была `Console.WriteLine()`, выводящая текст на консоль. При разработке приложений она очень удобна для получения дополнительной информации по выполняемым действиям:

```
Console.WriteLine("Вызывается функция MyFunc().");
MyFunc("Какие-то действия.");
Console.WriteLine("Функция MyFunc() завершена.");
```

Этот фрагмент кода показывает, как можно получать дополнительную информацию о функции с именем `MyFunc()`. Все это хорошо, но только окно консоли становится немного загроможденным, а при разработке приложений других типов, например, Windows Forms, консоль для вывода такой информации вообще недоступна. Но имеется и другой вариант — вывод текста в специальное место, а именно, в предлагаемое IDE-средой окно Output (Вывод).

В главе 2, где описывалось окно Error List (Список ошибок), было сказано, что в этом месте могут отображаться и другие окна. Одним из них является окно Output, которое может быть очень полезным для отладки. Для его вывода нужно выбрать в меню View (Вид) пункт Output (Вывод). Это окно содержит информацию о компиляции и выполнении кода,

включая ошибки, обнаруженные при компиляции. Еще его можно использовать, как показано на рис. 7.1, для отображения специальной диагностической информации.

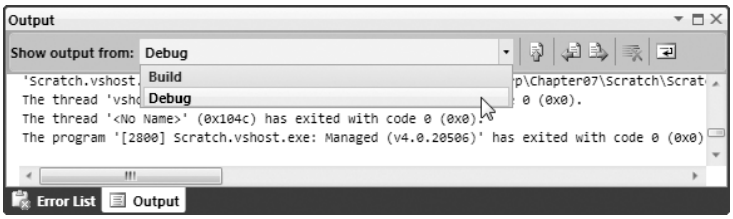


Рис. 7.1. *Окно* Output

В окне Output *имеется раскрывающееся меню, из которого можно выбирать разные режимы, в том числе и* Build *(Сборка) и* Debug *(Отладка). В этих режимах отображается информация соответственно только о компиляции или только о выполнении кода. Под "выводом в окно* Output*" в настоящем разделе подразумевается "вывод в представление режима* Debug *окна* Output*".*

Можно сделать и по-другому: создать файл журнала и при выполнении приложения фиксировать в нем всю важную информацию. Для этого применяется примерно такая же техника, как и для записи в окно Output, но нужно разбираться в работе с файловой системой из C#-приложений. Поэтому мы пока отложим рассмотрение этой темы, т.к. есть много возможностей и без возни с файлами.

Вывод отладочной информации

Выводить текст в окно Output во время выполнения очень просто. Нужно лишь вместо вызовов `Console.WriteLine()` использовать вызовы, которые выводят текст в нужное место:

- `Debug.WriteLine()`
- `Trace.WriteLine()`

Эти команды функционируют практически одинаково с одним важным отличием: первая работает только в отладочных сборках, а вторая — и в рабочих сборках тоже. На самом деле команда `Debug.WriteLine()` даже не компилируется в рабочую сборку; она просто исчезает, что, несомненно, имеет свои преимущества (в скомпилированном коде будет на одну команду меньше). В сущности, из одного исходного файла можно создавать две версии приложения. Отладочный вариант выводит различные дополнительные диагностические сведения, а в рабочем этих расходов не будет, и пользователей не будут раздражать отладочные сообщения.

Эти функции работают не совсем так, как `Console.WriteLine()`. Они принимают только один строковый параметр для выводимого сообщения и не позволяют вставлять значения переменных с помощью синтаксиса `{X}`. Это означает, что нужно дополнительно заботиться о вставке значений переменных в строки — например, с помощью операции конкатенации +. Еще (при желании) можно передавать второй строковый параметр с категорией выводимого текста. Тогда сразу видно, какие выходные сообщения отображаются в окне Output — это может оказаться удобным в случае вывода многих похожих сообщений из разных мест приложения.

В общем случае вывод этих функций выглядит так:

<категория>: *<сообщение>*

Например, оператор, в котором в необязательном параметре категории передается MyFunc:

```
Debug.WriteLine("Увеличение i на 1", "MyFunc");
```

приведет к выводу такой строки:

```
MyFunc: Увеличение i на 1
```

В следующем практическом занятии демонстрируется пример вывода отладочной информации.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Вывод текста в окно Output

1. Создайте новое консольное приложение с именем Ch07Ex01 и сохраните его в каталоге C:\BegVCSharp\Chapter07.

2. Измените его код следующим образом:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace Ch07Ex01
{
   class Program
   {
      static void Main(string[] args)
      {
        int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};
        int[] maxValIndices;
        int maxVal = Maxima(testArray, out maxValIndices);
        Console.WriteLine("Maximum value {0} found at element indices:", maxVal);
                          // Вывод максимального значения и его индексов
        foreach (int index in maxValIndices)
        {
           Console.WriteLine(index);
        }
        Console.ReadKey();
      }
      static int Maxima(int[] integers, out int[] indices)
      {
        Debug.WriteLine("Maximum value search started.");
                        // Начало поиска максимального значения
        indices = new int[1];
        int maxVal = integers[0];
        indices[0] = 0;
        int count = 1;
        Debug.WriteLine(string.Format(
          "Maximum value initialized to {0}, at element index 0.", maxVal));
        // Инициализация максимального значения элементом с индексом 0
        for (int i = 1; i < integers.Length; i++)
        {
          Debug.WriteLine(string.Format(
             "Now looking at element at index {0}.", i));
           // Просмотр элемента с индексом i
          if (integers[i] > maxVal)
          {
            maxVal = integers[i];
```

```
            count = 1;
            indices = new int[1];
            indices[0] = i;
            Debug.WriteLine(string.Format(
               "New maximum found. New value is {0}, at element index {1}.",
             // Найдено новое максимальное значение
               maxVal, i));
         }
         else
         {
            if (integers[i] == maxVal)
            {
               count++;
               int[] oldIndices = indices;
               indices = new int[count];
               oldIndices.CopyTo(indices, 0);
               indices[count - 1] = i;
               Debug.WriteLine(string.Format(
                  "Duplicate maximum found at element index {0}.", i));
                // Найден повтор максимального значения
            }
         }
      }
      Trace.WriteLine(string.Format(
         "Maximum value {0} found, with {1} occurrences.", maxVal, count));
       // Вывод максимального значения и количества его вхождений
      Debug.WriteLine("Maximum value search completed.");
                     // Поиск максимальных значений завершен
      return maxVal;
   }
  }
}
```

Фрагмент кода **Ch07Ex01\Program.cs**

3. Запустите приложение в режиме отладки. На рис. 7.2 показан результат, который должен получиться.

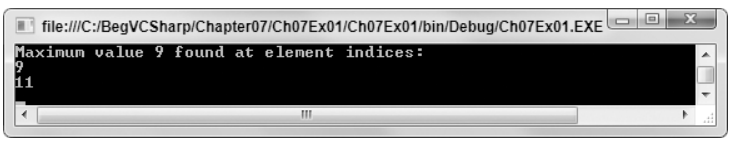


Рис. 7.2. *Приложение* `Ch07Ex01` *в действии*

4. Завершите работу этого приложения и просмотрите содержимое окна Output (Вывод) в отладочном режиме. Ниже приведена часть вывода:

```
...
Maximum value search started.
Maximum value initialized to 4, at element index 0.
Now looking at element at index 1.
New maximum found. New value is 7, at element index 1.
Now looking at element at index 2.
Now looking at element at index 3.
Now looking at element at index 4.
Duplicate maximum found at element index 4.
Now looking at element at index 5.
Now looking at element at index 6.
Duplicate maximum found at element index 6.
```

```
Now looking at element at index 7.
New maximum found. New value is 8, at element index 7.
Now looking at element at index 8.
Now looking at element at index 9.
New maximum found. New value is 9, at element index 9.
Now looking at element at index 10.
Now looking at element at index 11.
Duplicate maximum found at element index 11.
Maximum value 9 found, with 2 occurrences.
Maximum value search completed.
The thread '<No Name>' (0xcc8) has exited with code 0 (0x0).
The thread '<No Name>' (0xccc) has exited with code 0 (0x0).
The program '[3648] Ch07Ex01.vshost.exe: Managed' has exited with code 0 (0x0).
Поток '<Без имени>' (0xcc8) завершен с кодом 0 (0x0).
Поток '<Без имени>' (0xccc) завершен с кодом 0 (0x0).
Программа '[3648] Ch07Ex01.vshost.exe: Managed' завершена с кодом 0 (0x0).
```

5. Переключитесь в рабочий режим, воспользовавшись раскрывающимся меню в стандартной панели инструментов, как показано на рис. 7.3.

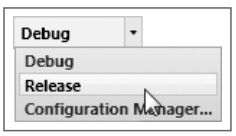


Рис. 7.3. *Переключение в рабочий режим*

6. Запустите программу снова, на этот раз в рабочем режиме, и опять загляните в окно Output, когда ее выполнение завершится. Ниже приведена часть вывода, который будет там содержаться:

```
...
Maximum value 9 found, with 2 occurrences.
The thread '<No Name>' (0xe2c) has exited with code 0 (0x0).
The thread '<No Name>' (0xd2c) has exited with code 0 (0x0).
The program '[1736] Ch07Ex01.vshost.exe: Managed' has exited with code 0 (0x0).
Поток '<Без имени>' (0xcc8) завершен с кодом 0 (0x0).
Поток '<Без имени>' (0xccc) завершен с кодом 0 (0x0).
Программа '[3648] Ch07Ex01.vshost.exe: Managed' завершена с кодом 0 (0x0).
```

Описание работы

Данное приложение является расширенным вариантом приложения из главы 6 и использует функцию для вычисления максимального значения в массиве целых чисел. Кроме того, оно возвращает массив индексов, указывающих места максимальных значений в массиве, чтобы вызывающий код мог работать с этими элементами.

В начале кода присутствует дополнительная директива:

```
using System.Diagnostics;
```

Она упрощает доступ к описанным выше функциям, поскольку они содержатся в пространстве имен `System.Diagnostics`. Без этой директивы `using` код вроде:

```
Debug.WriteLine("Bananas");
```

требовал бы дополнительного уточнения:

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

Так что директива `using` упрощает и сокращает код.

В коде функции `Main()` инициализируется тестовый массив целых чисел с именем `testArray`, а также объявляется еще один целочисленный массив по имени `maxValIndices` (для хранения индексов, возвращаемых функцией `Maxima()`), и затем выполняется вызов функции `Maxima()`. После возврата из функции код просто выводит результаты.

Функция `Maxima()` выглядит немного сложнее, но никакого непривычного для вас кода в ней нет. Поиск по массиву выполняется практически так же, как и в функции `MaxVal()` из главы 6, но только с сохранением индексов максимальных значений.

Обратите особое внимание (помимо строк с отладочной информацией) на способ сохранения индексов. Функция `Maxima()` возвращает не массив с размером, достаточным для хранения всех индексов в исходном массиве (т.е. с таким же размером, как у исходного массива), а массив с размером, достаточным для хранения найденных в данный момент индексов. Для этого она в ходе поиска постоянно заново создает массивы разного размера; это необходимо из-за того, что изменять размер массивов после создания нельзя.

Поиск начинается с предположения, что первый элемент в исходном массиве (с локальным именем `integers`) является максимальным значением, и что в массиве есть только одно максимальное значение. Это позволяет установить первоначальные значения для параметра `maxVal` (максимальное значение, возвращаемое функцией) и для выходного параметра массива `indices`, в котором сохраняются индексы обнаруженных максимальных значений. Поэтому в `maxVal` заносится значение первого элемента в `integers`, а в `indices` — единственное значение `0`, т.е. индекс первого элемента в массиве. Кроме того, количество обнаруженных максимальных значений сохраняется в переменной `count` — это нужно для работы с массивом `indices`.

Основное тело функции составляет цикл, в котором выполняется проход по всем значениям в массиве `integers`, кроме первого, т.к. оно уже было обработано. Каждое значение сравнивается с текущим значением `maxVal` и игнорируется, если `maxVal` больше. Если же это значение элемента массива оказывается больше `maxVal`, тогда соответственно изменяются значения `maxVal` и `indices`. Если же оно равно `maxVal`, то происходит увеличение значения переменной `count`, а для `indices` создается новый массив. Этот новый массив содержит на один элемент больше прежнего массива `indices`, среди которых находится и новый обнаруженный индекс.

Вот код, выполняющий все эти действия:

```
if (integers[i] == maxVal)
{
  count++;
  int[] oldIndices = indices;
  indices = new int[count];
  oldIndices.CopyTo(indices, 0);
  indices[count - 1] = i;
  Debug.WriteLine(string.Format(
    "Duplicate maximum found at element index {0}.", i));
}
```

Здесь резервная копия старого массива `indices` сохраняется в целочисленном массиве `oldIndices`, локальном по отношения к данному блоку `if`. Обратите внимание, что значения в `oldIndices` копируются в новый массив `indices` с помощью функции *<массив>*`.CopyTo()`. Эта функция принимает массив назначения и индекс первого копируемого элемента и копирует все значения в этот массив.

В коде многократно выполняется вывод различных текстовых строк с помощью функций `Debug.WriteLine()` и `Trace.WriteLine()`. Для вставки в эти строки значений переменных, как и в случае использования `Console.WriteLine()`, применяется функция `string.Format()`. Это немного эффективнее, чем операция конкатенации `+`.

При запуске приложения в отладочном режиме отображается список всех шагов, которые выполняются для получения результата. В рабочем режиме отображается только результат вычисления, поскольку в рабочих сборках вызовы функции `Debug.WriteLine()` не выполняются.

Помимо этих функций `WriteLine()` существует еще несколько, о которых следует знать. Во-первых, доступны эквиваленты функции `Console.Write()`:

- `Debug.Write()`
- `Trace.Write()`

Обе эти функции используют точно такой же синтаксис, как и функции `WriteLine()` — т.е. принимают один или два параметра с выводимым сообщением и необязательной категорией — но отличаются тем, что не добавляют символы конца строки.

Кроме того, имеются следующие команды:

- `Debug.WriteLineIf()`
- `Trace.WriteLineIf()`
- `Debug.WriteIf()`
- `Trace.WriteIf()`

Каждая из них имеет те же самые параметры, что и аналог без `If`, плюс один дополнительный обязательный параметр, который предшествует всем остальным в списке. Этот параметр принимает логическое значение (или выражение), и функция выводит текст только в том случае, если это значение равно `true`. Такие функции можно применять для условного вывода текста в окно Output.

Например, для вывода отладочной информации только при определенном условии можно использовать в коде ряд операторов `Debug.WriteLineIf()`, которые зависят от этого условия. В случае невыполнения условия операторы не будут выводить данные, и окно Output не будет загромождено излишней информацией.

Точки трассировки

Кроме вывода информации в окно Output, можно использовать точки трассировки (tracepoint). Они являются средством VS, а не языка C#, но служат для той же цели, что и функция `Debug.WriteLine()`. По сути, они позволяют выводить отладочную информацию без внесения изменений в код.

Точки трассировки доступны только в VS, в VCE их нет. Поэтому те, кто работают с VCE, могут пропустить этот раздел.

Чтобы ознакомиться с точками трассировки, мы вставим их на место отладочных команд, которые использовались в предыдущем примере. (См. файл `Ch07Ex01TracePoints` в коде для этой главы.) Ниже перечислены шаги для добавления точки трассировки.

1. Поместите курсор на строку, где нужно вставить точку трассировки. Точка трассировки будет обрабатываться *перед* выполнением этой строки кода.
2. Щелкните правой кнопкой мыши на строке кода и выберите в открывшемся контекстном меню пункт Breakpoint⇨Insert Tracepoint (Точка останова⇨Вставить точку трассировки).
3. Введите подлежащую выводу строку в текстовом поле Print a Message (Вывести сообщение) появившегося диалогового окна When Breakpoint Is Hit (При достижении точки останова). Если на экран необходимо вывести значение какой-то переменной, укажите имя этой переменной в фигурных скобках.
4. Щелкните на кнопке OK. Слева от строки кода, содержащей точку трассировку, появится красный ромбик, а сама строка будет выделена красным фоном.

По названию диалогового окна, в котором добавляются точки трассировки, и пунктов меню для их добавления понятно, что точки трассировки являются разновидностью точек останова (и при необходимости могут, как и точки останова, приостановить выполнение приложения). О точках останова, которые обычно применяются для выполнения более сложных процедур отладки, речь пойдет позже в этой главе.

На рис. 7.4 показана точка трассировки для строки 31 в примере `Ch07Ex01TracePoints`, в коде которого строки пронумерованы после удаления существующих операторов `Debug.WriteLine()`.

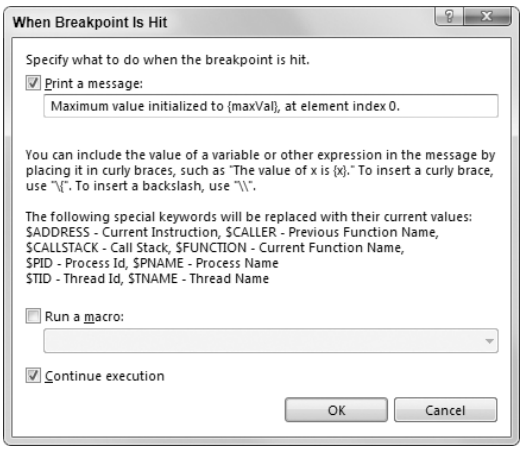


Рис. 7.4. *Пример добавления точки трассировки*

Как видно на рис. 7.4, точки трассировки позволяют вставлять и другую полезную информацию, связанную с их местоположением и контекстом. Поэкспериментируйте с этими значениями, а особенно с `$FUNCTION` и `$CALLER`, чтобы увидеть, какую дополнительную информацию можно собирать с их помощью. Кроме того, на рисунке видно, что точки трассировки могут выполнять макросы — более сложное средство, которое здесь не рассматривается.

В VS доступно еще одно окно — для быстрого просмотра точек трассировки в приложении. Оно открывается выбором пункта меню Debug⇨Windows⇨Breakpoints (Отладка⇨Окна⇨Точки останова) и представляет собой общее окно для отображения точек останова (разновидностью которых, как уже было сказано, являются точки трассировки). Его можно настроить так, чтобы в нем было больше сведений о точках трассировки, добавив из раскрывающегося меню Columns (Столбцы) столбец When Hit (При достижении). На рис. 7.5 показано это окно с добавленным столбцом When Hit и всеми вставленными в `Ch07Ex01TracePoints` точками трассировки.

Выполнение этого приложения в отладочном режиме приведет к получению тех же результатов, что и ранее. Можно удалять или временно отключать точки трассировки, щелкая на них правой кнопкой мыши в окне кода, либо с помощью окна Breakpoints. Флажок слева от точки трассировки в окне Breakpoints указывает, что она активна; отключенные точки трассировки не имеют таких пометок и отмечаются в окне кода контурным, а не сплошным ромбиком.

Сравнение вывода диагностической информации и точек трассировки

Теперь, после знакомства с двумя разными методиками для вывода практически одинаковой информации, давайте рассмотрим преимущества и недостатки каждой из них. У точек трассировки нет никаких эквивалентных команд `Trace()`, т.е. выводить информацию с использованием точек трассировки в рабочих сборках нельзя. Объясняется это тем, что точки трассировки не включаются в состав приложения. Они обрабатываются Visual Studio, и потому их нет в скомпилированной версии приложения. Они существуют только во время выполнения приложения в отладчике VS.

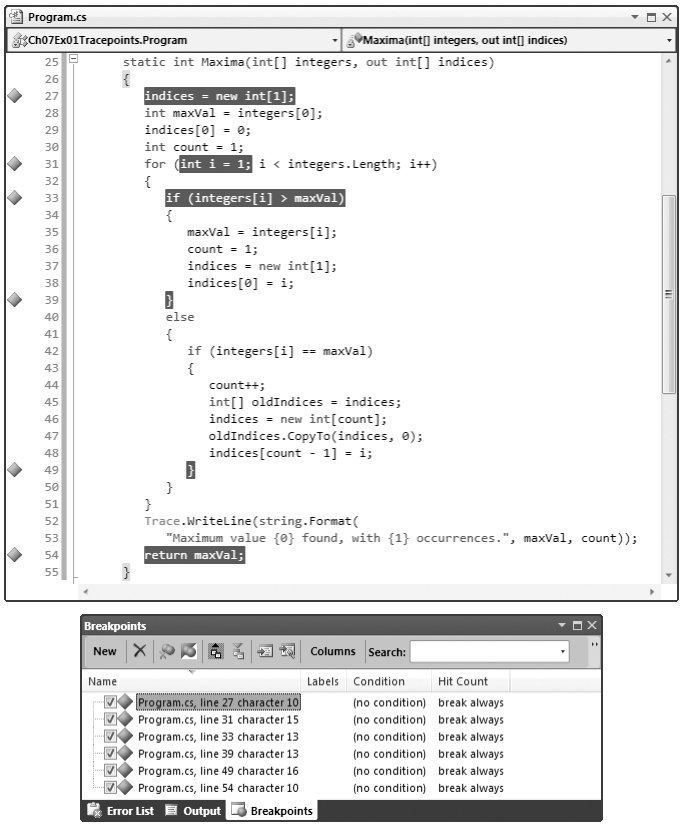


Рис. 7.5. *Окно* Breakpoints *с добавленным столбцом* When Hit

Главный недостаток точек трассировки является и их главным преимуществом: они хранятся в VS. Это позволяет быстро и легко добавлять их в приложения, когда они необходимы, а также без труда удалять. Для удаления точки трассировки достаточно щелкнуть на соответствующем красном ромбике — если она выводит слишком много излишней информации.

А одним из преимуществ точек трассировки является легкость добавления дополнительной информации, вроде упоминавшихся в предыдущем разделе значений `$FUNCTION`. Эта информация доступна и для кода, который пишется с использованием команд `Debug` и `Trace`, но там получать ее сложнее. Ниже приведены общие рекомендации по применению этих двух способов вывода отладочной информации.

- **Вывод диагностической информации**. Лучше применять, когда нужно, чтобы отладочные данные всегда выводились из приложения, особенно если выводимые строки сложны и содержат несколько переменных или много информации. Кроме того, команды `Trace` зачастую оказываются единственно возможным вариантом при необходимости в выводе информации во время выполнения приложения в рабочем режиме.
- **Точки трассировки**. Их лучше применять при отладке приложения для быстрого вывода важной информации, которая может помочь в исправлении семантических ошибок.

Есть еще одно очевидное отличие: точки трассировки доступны только в VS, а вывод диагностической информации можно выполнять как в VS, так и в VCE.

Отладка в режиме останова

Остальные приемы отладки, описываемые в настоящей главе, работают в режиме останова. В этот режим можно перейти несколькими способами, и все они, так или иначе, приостанавливают выполнение программы.

Переход в режим останова

Простейшим способом перехода в режим останова является щелчок на кнопке паузы в IDE-среде во время выполнения приложения. Эта кнопка находится в панели инструментов Debug (Отладка), которую необходимо специально добавить к панелям инструментов, видимым в VS по умолчанию. Для этого нужно щелкнуть правой кнопкой мыши в области панелей инструментов и выбрать в контекстном меню пункт Debug (Отладка) — появится панель Debug (рис. 7.6).

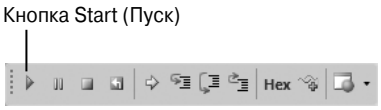


Рис. 7.6. *Внешний вид панели* Debug

Первые четыре кнопки в этой панели инструментов позволяют вручную управлять остановом. На рис. 7.6 три из них не активны, поскольку не работают с программой, которая еще не запущена. Активная кнопка, Start, идентична кнопке со стандартной панели инструментов. Остальные кнопки будут описаны далее по мере необходимости.

При запуске приложения эта панель инструментов приобретает вид, показанный на рис. 7.7.

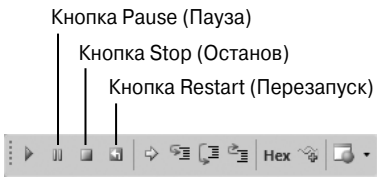


Рис. 7.7. *Внешний вид панели* Debug *после запуска приложения*

После запуска приложения три следующих за Start кнопки становятся активными и позволяют сделать следующее:

- приостановить приложение и перейти в режим останова;
- полностью остановить приложение (без перехода в режим останова, т.е. просто выйти из приложения).
- запустить приложение заново.

Приостановка приложения — пожалуй, простейший способ перехода в режим останова, но она не позволяет выбрать точное место останова. Остановка наверняка произойдет в наиболее пригодном для этого месте в приложении, например, в момент запроса данных от пользователя. Возможно, останов будет выполнен во время выполнения длительной операции или длинного цикла, но точный момент все равно будет довольно случайным. Обычно гораздо лучше вместо этого использовать точки останова.

Точки останова

Точка останова (breakpoint) представляет собой отметку в исходном коде, которая приводит к автоматическому переходу в режим останова. Точки останова доступны и в VS, и VCE, но в VS имеют больше возможностей. Их можно настраивать для выполнения перечисленных ниже действий.

- Переходить в режим останова при достижении точки останова.
- (Только в VS.) Переходить в режим останова при достижении точки останова только в случае, если указанное логическое выражение равно `true`.
- (Только в VS.) Переходить в режим останова при достижении точки останова определенное количество раз.
- (Только в VS.) Переходить в режим останова при достижении точки останова в случае, если с момента последнего достижения точки останова значение переменной изменилось.
- (Только в VS.) Выводить текст в окно Output или выполнять макрос (об этом уже упоминалось ранее в разделе "Точки трассировки").

Выполнение всех этих действий возможно только в отладочных сборках. При компиляции рабочей сборки все точки останова игнорируются.

Для добавления точек останова существует несколько способов. Для добавления простой точки останова, приостанавливающей приложение при достижении строки, можно щелкнуть левой кнопкой мыши слева от этой строки, либо щелкнуть правой кнопкой мыши на этой строке и выбрать в контекстном меню пункт Breakpoint⇨Insert Breakpoint (Точка останова⇨Вставить точку останова), либо выбрать пункт меню Debug⇨Toggle Breakpoint (Отладка⇨Включение/отключение точки останова), либо нажать клавишу <F9>.

После этого точка останова обозначается красным кружком рядом с выделенной цветом строкой кода, как показано на рис. 7.8.

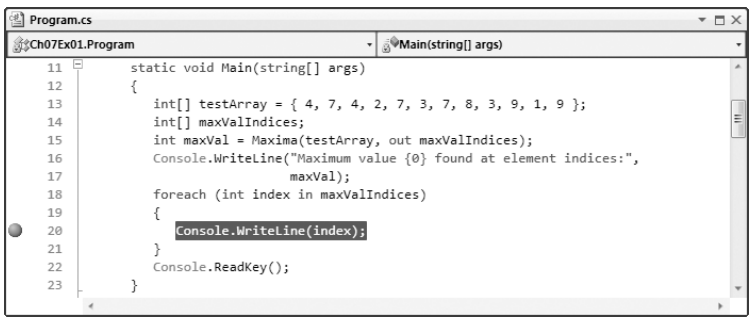


Рис. 7.8. *Отображение точки останова*

Материал, излагаемый в остальной части настоящего раздела, касается только VS, но не VCE. Поэтому те, кто использует VCE, могут сразу перейти к следующему разделу — "Другие способы перехода в режим останова".

В VS можно просматривать информацию о точках останова с помощью окна Breakpoints (о том, как его открыть, уже рассказывалось ранее в этой главе). Это окно позволяет также отключать точки останова (снимая отметки слева от их описания; отключенные точки останова помечаются незакрашенными красными кружками), удалять точки останова и редактировать их свойства.

Столбцы Condition (Условие) и Hit Count (Количество прохождений), видимые в этом окне по умолчанию, самые полезные из всех возможных столбцов. Редактировать содер-

жащиеся в них свойства можно, щелкнув правой кнопкой мыши на требуемой точке (либо в этом окне, либо непосредственно в окне кода) и выбрав в контекстном меню, соответственно, пункт Condition (Условие) или пункт Hit Count (Количество прохождений).

После выбора пункта Condition открывается диалоговое окно, в котором можно ввести любое логическое выражение с использованием любых переменных из области видимости данной точки останова. Например, можно задать такую точку останова, которая срабатывает при ее достижении, только если значение `maxVal` больше 4 — для этого нужно ввести выражение `maxval > 4` и выбрать вариант `IsTrue`. Можно также проверять значение выражения и активировать точку останова только при его изменении (например, при изменении `maxval` с 2 на 6).

После выбора пункта Hit Count открывается диалоговое окно, в котором можно задать количество проходов точки останова, после которого она должна срабатывать. В раскрывающемся списке имеются следующие варианты:

- Break always (Безусловный останов);
- Break when the hit count is equal to (Останов, когда количество прохождений равно);
- Break when the hit count is a multiple of (Останов, когда количество проходов кратно);
- Break when the hit count is greater than or equal to (Останов, когда количество проходов больше или равно).

Выбранный в этом списке вариант и значение, указанное в текстовом поле напротив списка, определяют поведение точки останова. Указывать количество проходов удобно в длинных циклах, в которых может потребоваться приостановить выполнение, скажем, после первых 5000 итераций. Без этого пришлось бы переходить в режим останова и возобновлять выполнение 5000 раз!

Точка останова с заданными дополнительными свойствами (вроде условия или количества прохождений) отображается по-другому. Вместо простого красного кружка такая точка останова отмечается красным кружком с белым плюсом внутри. Это позволяет сразу видеть, какие точки останова будут сразу же приостанавливать работу, а какие — только при определенных обстоятельствах.

Другие способы перехода в режим останова

Существуют еще два способа для перехода в режим останова. Первый выполняется при возникновении необрабатываемого исключения и будет описан позже в этой главе, при рассмотрении обработки ошибок. Второй способ выполняется при генерации *утверждения* (assertion).

Утверждениями называются такие инструкции, которые прерывают выполнение приложения с выдачей пользовательского сообщения. Они часто применяются во время разработки приложений для проверки работоспособности приложения. Например, в какой-то точке в приложении значение определенной переменной должно быть меньше 10. С помощью утверждения можно проверить, что это так, и прервать работу программы, если это не так. При срабатывании утверждения на выбор предлагается три действия: Abort (Прервать), которое завершает работу приложения; Retry (Повторить попытку), которое приводит к переходу в режим останова; и Ignore (Игнорировать), которое позволяет приложению продолжать работу обычным образом.

Как и рассмотренные ранее функции для вывода отладочной информации, функция утверждения доступна в двух вариантах:

- `Debug.Assert()`
- `Trace.Assert()`

Отладочная версия (Debug) компилируется только в отладочных сборках.

Эти функции принимают три параметра. Первый имеет тип bool, значение false которого приводит к срабатыванию утверждения. Второй и третий — строковые параметры, они отвечают за вывод информации во всплывающее диалоговое окно и окно Output. Для нашего предыдущего примера вызов функции должен выглядеть так:

```
Debug.Assert(myVar < 10, "myVar is 10 or greater.", "Assertion occurred in Main().");
                    // myVar больше или равно 10. В Main() сработало утверждение.
```

Утверждения часто удобны на ранних стадиях освоения пользователем приложения. Разработчик может включать в рабочую версию своего приложения вызовы Trace.Assert() для наблюдения за происходящим. В случае генерации утверждения пользователь получит уведомление, а соответствующая информация может быть отправлена разработчику — это поможет найти причину неверного действия приложения.

Например, в первом строковом параметре разработчик может передать краткое описание ошибки, а во втором — инструкции, что делать дальше:

```
Trace.Assert(myVar < 10, "Variable out of bounds.",
                         // Переменная вышла за допустимые пределы
          "Please contact vendor with the error code KCW001.");
        // Свяжитесь с поставщиком и сообщите о коде ошибки KCW001
```

При срабатывании этого утверждения пользователь увидит на экране диалоговое окно, показанное на рис. 7.9.

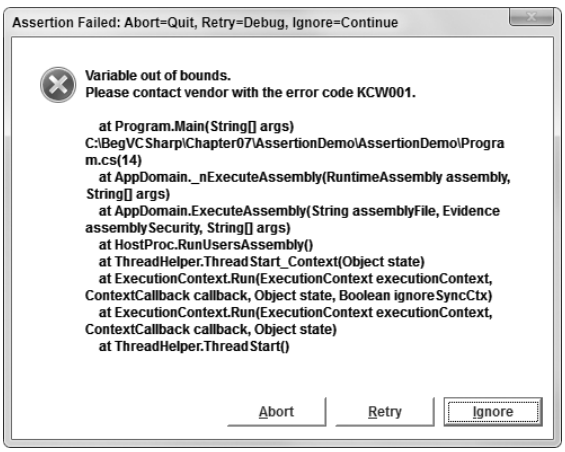


Рис. 7.9. *Диалоговое окно, отображаемое при срабатывании утверждения*

Конечно, такое окно не очень-то дружественно к пользователю, поскольку в нем содержится масса загадочной информации, но его снимок поможет разработчику быстро отыскать проблему.

Теперь пора узнать, что можно делать после остановки приложения и перехода в режим останова. Обычно переход в этот режим применяется для поиска ошибки в коде (или чтобы удостовериться, что все работает так, как надо). В режиме останова можно использовать самые разнообразные приемы, позволяющие анализировать код и точное состояние приложения на момент приостановки.

Наблюдение за содержимым переменных

Наблюдение за содержимым переменных — лишь один пример облегчения работы в VS и VCE. Проще всего проверить значение переменной так: навести курсор мыши на ее

имя в исходном коде, когда приложение находится в режиме останова. При этом появится желтая всплывающая подсказка с информацией о переменной, в том числе и с ее текущим значением.

Подобным образом можно выделять даже целые выражения и получать информацию об их результатах. Для составных значений вроде массивов можно разворачивать значения в подсказке, чтобы просматривать отдельные элементы.

Вы, видимо, заметили, что при выполнении приложения компоновка некоторых окон в IDE-среде изменяется. По умолчанию во время выполнения, как правило, происходят следующие изменения (это поведение может немного отличаться в зависимости от параметров конкретной установки).

- Окно Properties исчезает вместе с некоторыми другими окнами (возможно, включая окно Solution Explorer).
- Окно Error List заменятся двумя новыми окнами в нижней части окна IDE-среды.
- В этих новых окнах появляется несколько новых вкладок.

Новая компоновка показана на рис. 7.10. Ваше представление может выглядеть не совсем так — т.е. могут немного отличаться некоторые вкладки и окна — но функции, которые доступны в этих окнах и рассматриваются далее, будут в точности совпадать, а само представление можно настроить любым образом как с помощью пунктов меню View (Вид) и Debug⇨Windows (Отладка⇨Окна) в режиме останова, так и просто перетаскивая окна на экране.

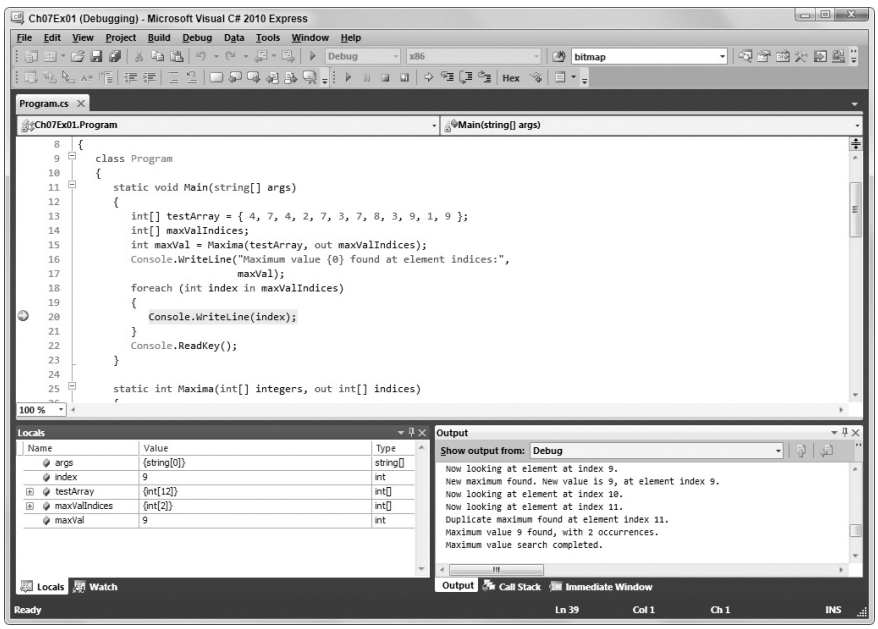


Рис. 7.10. *Внешний вид окна IDE-среды во время работы приложения*

Одно из новых окон, которое появляется в нижнем левом углу, особенно полезно для отладки. Оно позволяет наблюдать за значениями переменных в приложении в режиме останова. Доступные в нем вкладки в VS и VCE выглядят немного по-разному.

- Вкладка Autos (Автоматические). Доступна только в VS и содержит все переменные, которые используются в текущем и предшествующем операторах (<Ctrl+D>, <A>).

- Вкладка Locals (Локальные). Содержит все переменные из области видимости (<Ctrl+D>, <L>).
- Вкладка Watch N (Наблюдение N). Настраиваемое отображение переменных и выражений. Под N здесь подразумевается одна из четырех версий, которые доступны при выборе пункта меню Debug⇨Windows⇨Watch⇨Watch N (Отладка⇨Окна⇨Наблюдение⇨Наблюдение N).

Все эти вкладки работают более или менее одинаково, хотя и с различными дополнительными средствами, зависящими от их назначения. Каждая вкладка содержит список переменных с информацией об имени, значении и типе каждой из них. Для более сложных переменных, наподобие массивов, предусмотрены символы разворачивания и сворачивания дерева (+ и –), которые отображаются слева от их имени и открывают или скрывают древовидное представление их содержимого. Например, на рис. 7.11 показана вкладка Locals, полученная после размещения точки останова в коде. На вкладке видно развернутое представление одной из массивных переменных — `maxValIndices`.

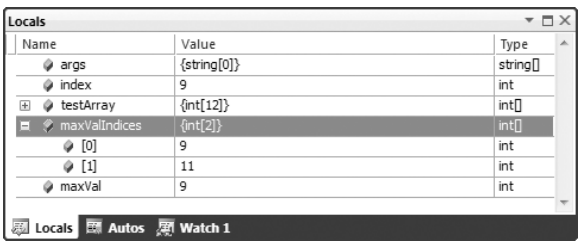


Рис. 7.11. *Вкладка* Locals *с развернутым представлением переменной* `maxValIndices`

Здесь также можно редактировать содержимое переменных. Это позволяет перекрыть любые другие операции присваивания значений переменным, выполненные ранее в коде. Для этого нужно просто ввести новое значение в столбце Value (Значение) для нужной переменной. Так можно, например, опробовать ситуации, для которых иначе потребовалось бы вносить изменения в код.

Окно Watch (или окна Watch, которых в VS может быть до четырех) позволяет наблюдать за конкретными переменными или за выражениями, содержащими конкретные переменные. Для этого введите в столбце Name (Имя) имя требуемой переменной или выражение и просмотрите результат. Учтите, что не все переменные в приложении постоянно находятся в области видимости — на это указывают пометки в окне Watch. Например, на рис. 7.12 показано окно Watch с несколькими переменными и выражениями, которое получено при срабатывании точки останова перед самым концом функции `Maxima()`.

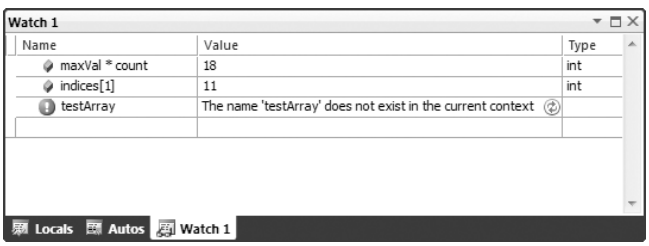


Рис. 7.12. *Пример окна* Watch

Массив `testArray` локален по отношению к `Main()`, поэтому отображается не его значение, а сообщение, что данная переменная в текущем контексте отсутствует.

Добавлять переменные в окно Watch можно и перетаскиванием из исходного кода.

У этого окна есть замечательная возможность: его различные представления позволяют видеть переменные, у которых изменились значения между прохождениями точек останова. Любое новое значение выделяется красным цветом, а не черным, и заметить изменившиеся значения значительно легче.

Как уже было сказано, для добавления дополнительных окон Watch в VS в режиме останова можно использовать варианты, доступные в меню Debug⇨Windows⇨Watch⇨Watch N, которые позволяют включать и отключать четыре возможных окна. В каждом из этих окон может содержаться отдельный набор наблюдаемых переменных и выражений, что дает возможность группировать взаимосвязанные переменные для облегчения доступа к ним.

Помимо этих окон, в VS имеется окно QuickWatch, которое содержит детальную информацию о любой переменной в исходном коде. Открыть его можно, щелкнув правой кнопкой мыши на нужной переменной и выбрав в контекстном меню пункт QuickWatch. Хотя обычно вполне достаточно стандартных окон Watch.

Критерии наблюдения сохраняются между запусками приложения. То есть после завершения и последующего запуска приложения снова добавлять наблюдаемые переменные и выражения не нужно: IDE-среда помнит, какие переменные и выражения просматривались в последний раз.

Пошаговое выполнение кода

Пока что было рассказано лишь о том, как узнавать, что происходит в приложениях после их перевода в режим останова. А теперь вы узнаете, как использовать IDE-среду для пошагового выполнения кода в режиме останова, что позволяет просматривать результаты выполнения каждой строки кода. Этот прием очень полезен при отладке приложений.

После перехода в режим останова слева от кода появляется курсор, который обычно сначала находится внутри красного кружка около точки останова, если она была причиной перехода в этот режим, и указывает на строку, которая должна выполняться следующей (рис. 7.13).

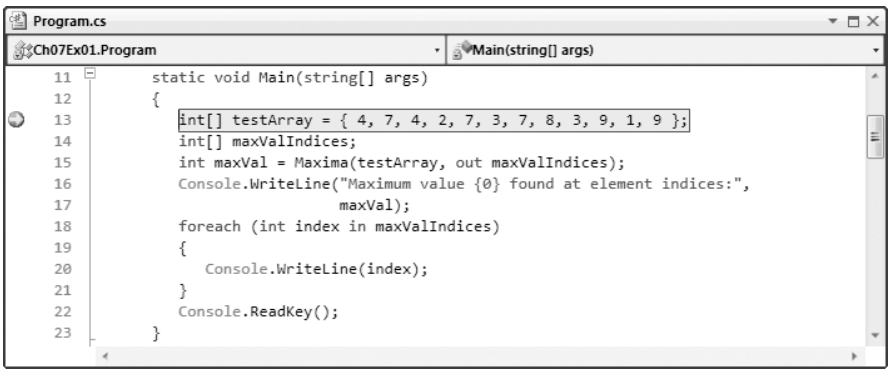


Рис. 7.13. *Курсор в режиме останова*

Он показывает, в каком месте выполнялся код, когда произошел переход в режим останова. С этого места выполнение кода можно продолжать построчно. Для этого используются кнопки панели инструментов Debug, показанные на рис. 7.14.

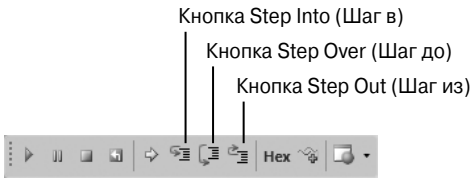


Рис. 7.14. *Кнопки панели инструментов* Debug*, которыми можно пользоваться для построчного выполнения кода*

Управление выполнением программы в режиме останова выполняется с помощью шестого, седьмого и восьмого значка панели Debug.

- Кнопка Step Into позволяет выполнить данный оператор и перейти к следующему выполняемому оператору.
- Кнопка Step Over похожа на кнопку Step Into, но пропускает вложенные блоки кода, в том числе и код функций.
- Кнопка Step Out позволяет перейти сразу к концу блока кода и возобновить режим останова со следующего за ним оператора.

Для просмотра каждой выполняемой приложением операции лучше использовать кнопку Step Into, которая выполняет инструкции одна за другой — даже по коду функций, вроде `Maxima()` из предыдущего примера. Если курсор находится на строке 15 (вызов функции `Maxima()`), то щелчок на этой кнопке приведет к переходу курсора на первую строку кода внутри `Maxima()`. А щелчок на кнопке Step Over при достижении строки 15 приведет к переходу курсора прямо на строку 16, без шагов внутри `Maxima()` (хотя этот код все равно выполняется). При входе в функцию, код которой не представляет никакого интереса, можно воспользоваться кнопкой Step Out для возврата в код после оператора вызова этой функции. По мере выполнения кода значения некоторых переменных будут меняться. Отследить эти изменения позволяют описанные выше окна наблюдения.

В коде, содержащем семантические ошибки, такой прием может оказаться самым полезным. Он позволяет перейти в коде к тому месту, которое предположительно является источником проблем, и все ошибки будут генерироваться, как и при обычном выполнении программы. Нужно просто при пошаговом выполнении следить за данными — и вы увидите причину проблемы. Позже в этой главе будет продемонстрирован пример применения этого приема.

Окна Immediate и Command

Окно Command (Команда; доступно только в VS) и окно Immediate (Немедленно; доступно через пункт меню Debug⇨Windows (Отладка⇨Окна)) позволяют выполнять команды во время работы приложения. Окно Command позволяет вручную выполнять в VS различные операции (т.е. операции, выбираемые в меню и панелях инструментов), а окно Immediate — выполнять дополнительный код, помимо исходного кода, а также вычислять выражения.

В VS оба эти окна тесно взаимосвязаны (в предыдущих версиях VS они вообще считались одним окном). Существуют даже команды переходов: команда `immed` для перехода из окна Command в окно Immediate, и команда `>cmd` для обратного перехода.

В настоящем разделе основное внимание уделяется окну Immediate, т.к. окно Command по-настоящему полезно только для выполнения сложных операций и к тому же доступно лишь в VS, а окно Immediate доступно и в VS, и в VCE. Проще всего это окно использовать для вычисления выражений — примерно как в окнах Watch. Для этого введите нужное выражение и нажмите клавишу <Enter>, и в окне появится соответствующая информация — см., например, рис. 7.15.

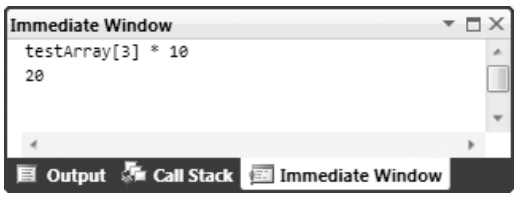


Рис. 7.15. *Вычисление выражения в окне* Immediate

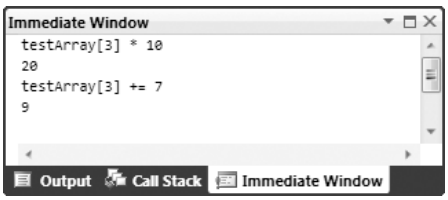


Рис. 7.16. *Изменение содержимого переменной в окне* Immediate

В этом окне также можно изменять содержимое переменных, как показано на рис. 7.16.

В большинстве случаев все это легче сделать с помощью описанных выше окон наблюдения переменных, но данное окно удобнее для настройки значений и для проверки выражений, результаты которых вряд ли будут интересны позже.

Окно Call Stack

Последнее окно, которое осталось рассмотреть, называется Call Stack (Стек вызовов) и показывает путь, которым управление дошло до текущего местоположения. Попросту говоря, это означает, что отображается текущая функция, функция, которая ее вызвала, функция, которая вызвала ту функцию, и т.д. (т.е. список вложенных вызовов функций). Отображаются и точные места вызовов этих функций.

В нашем примере при переходе в режим останова при выполнении функции `Maxima()` или пошаговом переходе внутрь этой функции в окне Call Stack появится информация, показанная на рис. 7.17.

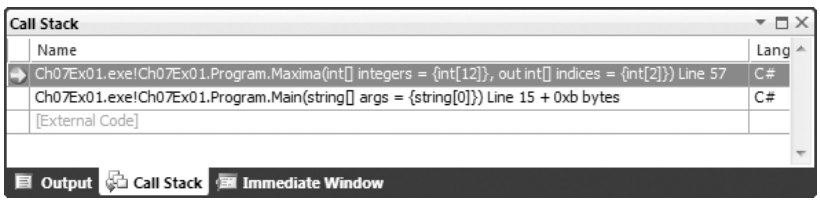


Рис. 7.17. *Пример информации, отображаемой в окне* Call Stack

Двойной щелчок на элементе позволяет перейти к соответствующему месту в коде и увидеть путь достижения в коде текущей точки. Данное окно особенно полезно при первоначальном обнаружении ошибки, поскольку позволяет увидеть ситуацию непосредственно перед возникновением этой ошибки. В случае возникновения ошибок в наиболее часто используемых функциях это позволяет определить их источник.

Иногда в окне Call Stack *выводится очень запутанная информация. Например, ошибки могут возникать и за пределами приложения, из-за неправильного применения внешних функций. В таких случаях в этом окне может содержаться длинная вереница элементов, в которой только один или два выглядят знакомо. Чтобы (при необходимости) просматривать внешние ссылки, щелкните в этом окне правой кнопкой мыши и выберите в контекстном меню пункт* Show External Code *(Показать внешний код).*

Обработка ошибок

В первой части настоящей главы рассказывалось о том, как можно отыскивать и исправлять ошибки во время разработки приложения, чтобы они не возникали в рабочей версии кода. Хотя, конечно, ошибки все равно возникают, и стопроцентного способа предотвратить их появление не существует. Поэтому лучше всегда ожидать возникновение проблем и писать код, достаточно устойчивый, чтобы аккуратно обрабатывать эти ошибки без прерывания процесса выполнения.

Все существующие для этого приемы вместе называются *обработкой ошибок*. В настоящем разделе речь пойдет об исключениях и способах их обработки. *Исключением* (exception) называется ошибка, которая генерируется в основном коде или в какой-то вызываемой им функции во время выполнения. Определение ошибки здесь немного более размыто, чем было до этого, поскольку исключения могут генерироваться вручную, внутри функций и т.д. Например, исключение внутри функции может генерироваться в случае, если один из ее строковых параметров не начинается с буквы "a". Строго говоря, вне контекста функции это вовсе не ошибка, но код будет трактовать ее именно как ошибку.

Вы уже встречались несколько раз с исключениями в этой книге. Пожалуй, простейшим примером является попытка обращения к элементу массива за его границами:

```
int[] myArray = {1, 2, 3, 4};
int myElem = myArray[4];
```

Этот код приведет к выводу сообщения `Index was outside the bounds of the array` (Индекс выходит за границы массива) и завершению приложения.

В этой книге уже были показаны окна, которые отображаются в подобных случаях. Они содержат информацию о вызвавшем проблему коде, ссылки для доступа к соответствующим материалам справочной системы .NET, а также ссылку View Detail (Подробнее), позволяющую получить дополнительные сведения о возникшем исключении.

Исключения определяются в пространствах имен и в большинстве случаев имеют имена, которые делают очевидным их назначение. В примере сгенерированное исключение имеет имя `System.IndexOutOfRangeException`, которое легко понять: предоставленный индекс выходит за пределы диапазона (out of range), допустимого в `myArray`. Появление сообщения об исключении и завершение работы приложения происходит, только если исключение никак не обрабатывается. В следующем разделе будет показано, что нужно делать для обработки исключения.

Конструкция `try...catch...finally`

В состав языка C# входит синтаксис для *структурной обработки исключений* (Structured Exception Handling — SEH). С помощью трех ключевых слов — `try`, `catch` и `finally` — код помечается как способный обрабатывать исключения и снабжается инструкциями, что следует делать при возникновении исключения. Все ключевые слова имеют связанный с ними блок кода и должны обязательно использоваться в строго следующем друг за другом порядке. Базовая структура выглядит так:

```
try
{
   ...
}
catch (<типИсключения> e)
{
   ...
}
```

```
finally
{
   ...
}
```

Можно создавать блоки `try` и `finally` без блока `catch` или блок `try` с несколькими блоками `catch`. При наличии одного и более блоков `catch` блок `finally` не обязателен, но при отсутствии блоков `catch` он должен присутствовать.

Способы применения всех этих блоков описаны ниже.

- **`try`**. Содержит код, который может генерировать (throw) исключения.
- **`catch`**. Содержит код, который должен выполняться при возникновении исключений. Параметры <*типИсключения*> позволяют настраивать блоки `catch` на реагирование только на исключения определенных типов (например, исключения типа `System.IndexOutOfRangeException`), поэтому можно указывать несколько блоков `catch`. Но можно и опустить этот параметр и получить универсальный блок `catch`, способный реагировать на исключения всех типов.
- **`finally`**. Содержит код, который выполняется всегда — либо после блока `try`, если не возникает никаких исключений, либо после блока `catch`, если обработано какое-то исключение, либо непосредственно перед возвратом по стеку вызовов из-за необработанного исключения. Это значит, что SEH позволяет создавать вложенные блоки `try...catch...finally` — или непосредственно, или с помощью вызова функции с блоком `try`. Например, если в вызванной функции возникло исключение, и оно не обработано никаким блоком `catch` внутри этой функции, то оно может быть обработано блоком `catch` из вызывающего кода. Но если подходящий блок `catch` так и не будет найден, то приложение завершится аварийно. Поэтому и нужен блок `finally`: иначе код можно было бы размещать и за пределами конструкции `try...catch...finally`. Такая вложенная обработка будет рассмотрена ниже, в разделе "Примечания по обработке исключений", так что не переживайте, если пока что-то непонятно.

Ниже приведена последовательность событий после возникновения исключения в коде внутри блока `try`.

- Выполнение блока `try` останавливается в точке, где возникло исключение.
- Если существует какой-нибудь блок `catch`, выполняется проверка, соответствует ли этот блок типу сгенерированного исключения. Если блока `catch` нет, выполняется блок `finally` (который, если нет ни одного блока `catch`, должен присутствовать обязательно).
- Если блок `catch` существует, но не тот, выполняется проверка, существуют ли другие подходящие блоки `catch`.
- Если имеется совпадающий с типом исключения блок `catch`, выполняется содержащийся в нем код, после чего выполняется блок `finally`, если он есть.
- Если не обнаружено ни одного совпадающего с типом исключения блока `catch`, выполняется блок кода `finally`, если он есть.

В следующем практическом занятии демонстрируется пример работы с исключениями — генерация исключений и несколько способов их обработки.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Обработка исключений

1. Создайте новое консольное приложение с именем `Ch07Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter07`.
2. Измените его код следующим образом (комментарии с номерами строк помогут сопоставить код с приведенным далее описанием):

```
class Program
{
   static string[] eTypes = {"none", "simple", "index", "nested index"};
   static void Main(string[] args)
   {
      foreach (string eType in eTypes)
      {
         try
         {
            Console.WriteLine("Main() try block reached.");        // Строка 23
                              // Достигнут блок try в Main()
            Console.WriteLine("ThrowException(\"{0}\") called.", eType); // 24
                              // Вызвано ThrowException
            ThrowException(eType);
            Console.WriteLine("Main() try block continues.");     // Строка 26
                              // Выполнение блока try в Main() продолжается
         }
         catch (System.IndexOutOfRangeException e)                // Строка 28
         {
            Console.WriteLine("Main() System.IndexOutOfRangeException catch"
                           + " block reached. Message:\n\"{0}\"",
                             // Достигнут блок catch ... Сообщение:
                           e.Message);
         }
         catch                                                    // Строка 34
         {
            Console.WriteLine("Main() general catch block reached.");
                             // Достигнут общий блок catch в Main()
         }
         finally
         {
            Console.WriteLine("Main() finally block reached.");
                             // Достигнут блок finally в Main()
         }
         Console.WriteLine();
      }
      Console.ReadKey();
   }
   static void ThrowException(string exceptionType)
   {
      Console.WriteLine("ThrowException(\"{0}\") reached.", exceptionType); // 49
                      // Вход в функцию ThrowException
      switch (exceptionType)
      {
         case "none" :
            Console.WriteLine("Not throwing an exception.");
                             // Исключение не генерируется
            break;                                                // Строка 54
         case "simple" :
            Console.WriteLine("Throwing System.Exception.");
                             // Генерируется исключение System.Exception
            throw (new System.Exception());                       // Строка 57
```

```
            case "index" :
              Console.WriteLine("Throwing System.IndexOutOfRangeException.");
                              // Генерируется System.IndexOutOfRangeException
              eTypes[4] = "error";                                    // Строка 60
              break;
            case "nested index":
              try                                                     // Строка 63
              {
                Console.WriteLine("ThrowException(\"nested index\") " +
                                  "try block reached.");
                        // Достигнут блок try в ThrowException ("вложенный индекс")
                Console.WriteLine("ThrowException(\"index\") called.");
                        // Вызвана функция ThrowException ("индекс")
                ThrowException("index");                              // Строка 68
              }
            catch                                                     // Строка 70
            {
              Console.WriteLine("ThrowException(\"nested index\") general"
                              + " catch block reached.");
             // Достигнут основной блок catch в ThrowException ("вложенный индекс")
            }
            finally
            {
              Console.WriteLine("ThrowException(\"nested index\") finally"
                              + " block reached.");
             // Достигнут блок finally в ThrowException ("вложенный индекс")
            }
            break;
          }
        }
    }
```

Фрагмент кода ***Ch07Ex02\Program.cs***

3. Запустите приложение. На рис. 7.18 показан результат, который должен получиться.

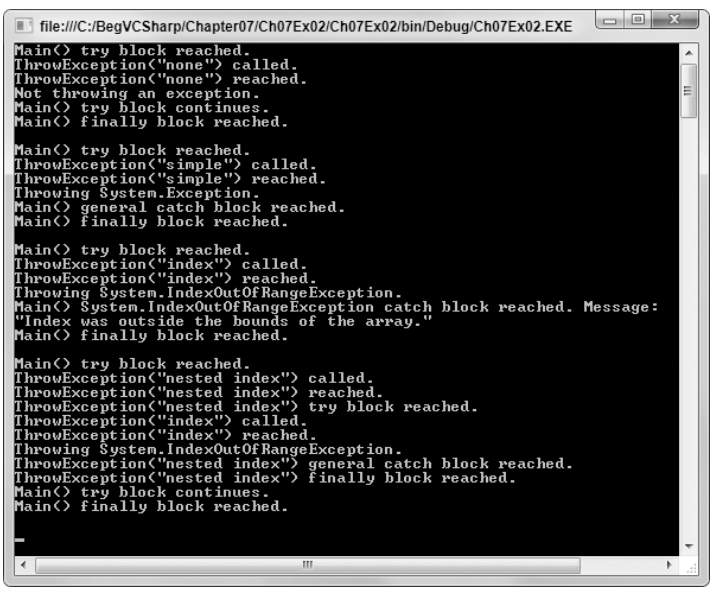


Рис. 7.18. *Приложение* `Ch07Ex02` *в действии*

Описание работы

У этого приложения в функции `Main()` имеется блок `try`, в котором вызывается функция `ThrowException()`. В зависимости от переданного ей параметра эта функция может генерировать различные исключения:

- `ThrowException("none")` — не генерирует исключение;
- `ThrowException("simple")` — генерирует общее исключение;
- `ThrowException("index")` — генерирует исключение `System.IndexOutOfRangeException`;
- `ThrowException("nested index")` — содержит собственный блок `try` с кодом, вызывающим `ThrowException("index")` для генерации исключения типа `System.IndexOutOfRangeException`.

Каждый из этих строковых параметров хранится в глобальном массиве `eTypes`, и все они перебираются в цикле внутри `Main()`, вызывая `ThrowException()` по одному разу с каждым параметром. При каждом проходе цикла на консоль выводятся различные сообщения для информирования о происходящем. Этот код предоставляет замечательную возможность опробовать описанные ранее в главе приемы пошагового выполнения кода. Построчное выполнение данного кода позволяет детально понять, как он функционирует.

Для этого добавьте новую точку останова (со стандартными свойствами) в 23-й строке кода, которая выглядит следующим образом:

```
Console.WriteLine("Main() try block reached.");
```

Здесь для работы с кодом используются номера строк, как они представлены в загружаемом коде. Если у вас нумерация строк отключена, включите ее, выбрав пункт меню Tools⇨Options (Сервис⇨Параметры), а затем перейдя в раздел Text Editor⇨C#⇨General (Текстовый редактор⇨C#⇨Общие). Комментарии с номерами включены в код, чтобы можно было следить за изложением, не открывая пронумерованный файл.

Запустите приложение в отладочном режиме. Практически сразу же программа перейдет в режим останова, а курсор будет находиться на строке 23. Перейдите на вкладку Locals (Локальные) окна наблюдения переменных, и вы увидите, что `eType` в настоящий момент имеет значение `"none"`. Воспользуйтесь кнопкой Step Into для выполнения строк 23 и 24 и удостоверьтесь, что первая строка текста действительно выведена на консоль. Затем снова воспользуйтесь кнопкой Step Into, чтобы войти в функцию `ThrowExpcetion()` — строка 25.

После перехода к функции `ThrowException()` (в строке 49) содержимое окна Locals изменится. `eType` и `args` уже не находятся в области видимости (т.к. локальны по отношению к `Main()`), а вместо них появляется локальный аргумент `exceptionType` — конечно, с тем же значением `"none"`. Продолжайте щелкать на кнопке Step Into до оператора `switch`, который проверит значение `exceptionType` и выведет строку `Not throwing an exception`. При выполнении оператора `break` (в строке 54) произойдет выход из кода функции и возобновление работы кода `Main()` в строке 26. Поскольку никакого исключения сгенерировано не было, продолжение работы кода будет выполняться в блоке `try`.

Затем обработка доходит до блока `finally`. Щелкните на кнопке Step Into еще несколько раз, чтобы пройти блок `finally` и завершить первую итерацию цикла `foreach`. При следующем выполнении строки 25 функция `ThrowExcetion()` будет вызвана уже с другим параметром — `simple`.

Продолжайте эксплуатировать кнопку Step Into, пока не достигнете строки 57:

```
throw (new System.Exception());
```

Здесь для генерации исключения используется ключевое слово `throw`. С ним нужно указать инициализированный с помощью оператора `new` параметр, и тогда оно сгенерирует соответствующее исключение. В данном примере вместе с ними используется еще одно исключение — `System.Exception` из пространства имен `System`.

При использовании оператора `throw` в блоке `case` оператор `break` не обязателен, т.к. `throw` также завершает выполнение блока.

При выполнении данного оператора с помощью кнопки Step Into управление будет передано общему блоку `catch`, который начинается в строке 34, т.к. соответствия с предыдущим блоком, начинающимся в строке 28, не обнаружено. Пошаговое выполнение этого кода проходит этот блок, блок `finally` и возвращается к выполнению еще одной итерации цикла, где в строке 25 производится вызов `ThrowException()` с новым параметром — на этот раз `"index"`.

Теперь `ThrowException()` сгенерирует исключение в строке 60:

```
eTypes[4] = "error";
```

Массив `eTypes` является глобальным, поэтому проблема здесь не в невозможности доступа к этому массиву, а в попытке обращения к его пятому элементу (ведь отсчет элементов начинается с 0), что приводит к генерации исключения `System.IndexOutOfRangeException`.

На этот раз в `Main()` имеется подходящий блок `catch`, и поэтому управление передается этому блоку, который начинается в строке 28. Вызов `Console.WriteLine()` в данном блоке выводит хранящееся в исключении сообщение с использованием `e.Message` (исключение доступно через параметр блока `catch`). Пошаговое выполнение снова проходит сначала через блок `finally` (а не второй блок `catch`, поскольку исключение уже обработано) и возвращается к очередной итерации цикла с вызовом `ThrowException()` в строке 25.

При достижении конструкции `switch` в `ThrowException()` управление войдет в новый блок `try`, начинающийся в строке 63. По достижении строки 48 будет выполнен вложенный вызов `ThrowException()`, на этот раз с параметром `"index"`. Можете воспользоваться кнопкой Step Over для пропуска уже знакомых строк кода. Как и ранее, данный вызов приведет к генерации исключения `System.IndexOutOfRangeException`, но теперь оно обрабатывается во вложенной структуре `try...catch...finally`, которая находится внутри `ThrowException()`. В этой структуре нет никакого явного совпадения для исключения такого типа, поэтому его обработкой займется код в общем блоке `catch` (который начинается в строке 70).

Как и при обработке предыдущих исключений, далее управление проходит этот блок `catch` и соответствующий блок `finally` и дойдет до конца вызова функции, но с одним важным отличием: исключение было не только сгенерировано, на и обработано — кодом в `ThrowException()`. А это значит, что исключения для обработки в `Main()` не останется, и поэтому далее управление передается непосредственно блоку `finally`, после чего работа приложения завершается.

Просмотр и конфигурирование исключений

В .NET Framework содержится множество исключений различного типа, любое из которых разработчик может генерировать, обрабатывать или даже генерировать в своем коде так, чтобы оно могло перехватываться в более сложных приложениях. В IDE-среде имеется специальное диалоговое окно для просмотра и редактирования доступных исключений, открыть которое можно, выбрав пункт меню Debug⇨Exceptions (Отладка⇨Исключения) или нажав клавиши <Ctrl+D>, <E>. Это диалоговое окно показано на рис. 7.19 (в VCE список содержит только второй и третий элементы).

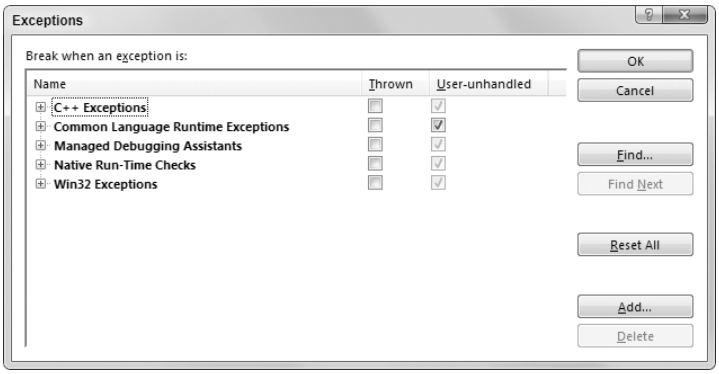


Рис. 7.19. *Диалоговое окно* Exceptions

Исключения в этом окне сгруппированы по категориям и пространствам имен библиотеки .NET. Например, чтобы увидеть исключения, доступные в пространстве имен `System`, можно развернуть сначала узел Common Language Runtime Exceptions (Исключения общеязыковой исполняющей среды), а затем узел System (Система). В открывшемся списке будет присутствовать и уже знакомое нам исключение `System.IndexOutOfRangeException`.

Каждое исключение можно конфигурировать с помощью флажков, которые расположены рядом. Первый флажок — Thrown (Генерировать) — позволяет выполнять прерывание и переход в режим отладки даже при наличии обработки исключений, а второй — чтобы необрабатываемые исключения игнорировались со всеми вытекающими из этого последствиями. В большинстве случаев игнорирование необрабатываемых исключений все равно приводит к переходу в режим останова, так что второй флажок применяется лишь в особых случаях.

Обычно вполне приемлемы стандартные настройки.

Примечания по обработке исключений

Блоки `catch` для конкретных исключений следует размещать перед более общим перехватом исключений — иначе приложение компилироваться не будет. Обратите также внимание, что генерировать исключения можно и внутри самих блоков `catch` — либо так же, как и в предыдущем примере, либо просто следующим выражением:

```
throw;
```

Этот оператор приводит к повторной генерации исключения, обрабатываемого блоком `catch`. Но в таком случае оно будет обрабатываться не текущим блоком `try...catch...finally`, а родительским кодом (хотя блок `finally` во вложенной структуре выполняться все-таки будет).

Например, если изменить блок `try...catch...finally` в `ThrowException()` из предыдущего примера так, как показано ниже:

```
try
{
  Console.WriteLine("ThrowException(\"nested index\") " + "try block reached.");
  Console.WriteLine("ThrowException(\"index\") called.");
  ThrowException("index");
}
catch
{
  Console.WriteLine("ThrowException(\"nested index\") general"
                    + " catch block reached.");
  throw;
}
```

```
finally
{
   Console.WriteLine("ThrowException(\"nested index\") finally"
                     + " block reached.");
}
```

то поток выполнения сначала перейдет сначала в показанный здесь блок `finally`, а затем в соответствующий блок `catch` в `Main()`. Результат в окне консоли будет выглядеть по-другому, как показано на рис. 7.20.

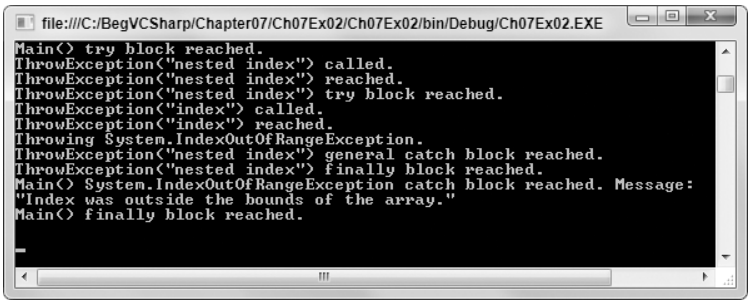


Рис. 7.20. *Вывод в окне консоли после изменения блока* `try...catch...finally`

На этом экранном снимке видны дополнительные строки, выведенные функцией `Main()`, поскольку теперь исключение `System.IndexOutOfRangeException` перехватывается в ней.

Резюме

В этой главе рассматривались приемы отладки приложений. Таких приемов много, и большинство из них годится для проектов любого типа, а не только консольных приложений.

Теперь вам известно обо всем, что может потребоваться для создания и отладки простых консольных приложений. Начиная со следующей главы, речь пойдет о мощной технологии объектно-ориентированного программирования.

Упражнения

1. "Лучше использовать функцию `Trace.WriteLine()`, а не `Debug.WriteLine()`, т.к. версия `Debug` работает только в отладочных сборках". Согласны ли вы с этим и почему?
2. Напишите код для простого приложения с циклом, которое генерирует ошибку после выполнения 5000 итераций. Используйте точку останова для перехода в режим останова непосредственно перед возникновением ошибки на 5000-й итерации. (Примечание: проще всего сгенерировать ошибку с помощью доступа к несуществующему элементу массива — например, к `myArray[1000]` в массиве из сотни элементов).
3. "Блоки кода `finally` выполняются только в том случае, если не выполняется блок `catch`". Так или нет?
4. С учетом приведенного ниже определения для типа перечисления `orientation` напишите приложение, использующее технологию структурной обработки исключений (SEH) для безопасного приведения переменной типа `byte` к `orientation`. (Примечание: принудительную генерацию исключений можно выполнить с помощью ключевого слова `checked`, как указано ниже в примере. Обязательно используйте этот код в приложении.)

```
enum orientation : byte
{
   north = 1,
   south = 2,
   east = 3,
   west = 4
}
myDirection = checked((orientation)myByte);
```

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Типы ошибок	Фатальные ошибки могут привести к полному краху приложения, как на этапе компиляции (синтаксические ошибки), так и во время выполнения. Семантические, или логические, ошибки менее очевидны и могут привести к неверной или непредсказуемой работе функции.
Вывод отладочной информации	В код можно включить вывод вспомогательной информации в окно Output для выполнения отладки с помощью IDE-среды. Для этого предназначено семейство функций `Debug` и `Trace` (`Debug` игнорируются в рабочих сборках). В производственных приложениях лучше записывать отладочные сообщения в файл журнала. В VS для вывода отладочной информации можно также использовать точки трассировки.
Режим останова	В режим останова (по сути, это состояние приостановки приложения) можно войти вручную, с помощью точек останова, с помощью утверждений или при возникновении необрабатываемого исключения. Точки останова можно размещать в коде где угодно, а в VS можно настроить точки останова так, чтобы они прерывали выполнение только при выполнении специальных условий. В режиме останова можно просматривать содержимое переменных (с помощью различных окон просмотра отладочной информации) и выполнять код построчно, что помогает понять, где могут находиться ошибки.
Исключения	Исключения — это ошибки, возникшие во время выполнения, которые можно перехватить и обработать программным образом, чтобы не дать приложению завершиться аварийно. Имеется много различных типов исключений, которые могут возникнуть при вызовах функций или при работе с переменными. Кроме того, можно генерировать исключения с помощью ключевого слова `throw`.
Обработка исключений	Исключения, которые не обрабатываются в коде, приводят к завершению работы приложения. Обработка исключений осуществляется с помощью блоков кода `try`, `catch` и `finally`. Блоки `try` содержат разделы кода, для которых выполняется обработка исключений. Блоки `catch` содержат код, который выполняется только при возникновении исключения, и могут соотноситься лишь с конкретными типами исключений. Блоков `catch` может быть несколько. Блоки `finally` содержат код, который выполняется после завершения обработки исключения или после завершения выполнения блока `try`, если исключение не возникло. Можно включить лишь один блок `finally`, но и он не обязателен при наличии хотя бы одного блока `catch`.

8

Введение в объектно-ориентированное программирование

В ЭТОЙ ГЛАВЕ...

- Что такое объектно-ориентированное программирование
- Приемы ООП
- Связь приложений Windows Forms с ООП

В предыдущих разделах был изложен весь базовый материал по синтаксису языка C# и программированию на нем, а также показано, как отлаживать приложения и создавать полезные консольные приложения. Однако для доступа ко всей мощи C# и .NET Framework необходимо использовать *объектно-ориентированное программирование* (ООП). Вообще-то, как скоро станет понятно, вы уже немного знакомы с этой техникой, но для простоты мы не акцентировали на этом внимание.

В настоящей главе кодирование временно откладывается, а все внимание переносится на принципы, лежащие в основе объектно-ориентированного программирования. Вы увидите, что C# тесно связан с ООП. Все понятия, вводимые в этой главе, будут более подробно рассматриваться в последующих главах на конкретных примерах, так что не переживайте, если в каком-то материале не разберетесь сразу.

Сначала будут рассмотрены основы ООП и, конечно же, ответ на самый фундаментальный вопрос: что такое *объект*. Терминология ООП поначалу может показаться несколько запутанной, поэтому предоставляется много объяснений. Вы также увидите, что использование ООП требует другого взгляда на программирование.

Помимо общих принципов ООП, в главе затрагивается и область, которая требует глубокого понимания ООП — приложения Windows Forms. Такие приложения (работающие в среде Windows и содержащие меню, кнопки и т.п.) предоставляют огромную область для описания и позволяют продемонстрировать основные моменты ООП в среде Windows Forms.

Концепции объектно-ориентированного программирования, описанные в данной главе, обусловлены средой .NET, и некоторые из приводимых здесь приемов могут не работать в других объектно-ориентированных средах. При программировании на C# применяются приемы ООП из .NET, поэтому мы будем рассматривать именно их.

Что такое объектно-ориентированное программирование

Объектно-ориентированное программирование является относительно новым подходом к созданию компьютерных приложений, который призван устранить многие из проблем, существующих в традиционных методиках программирования. Вид программирования, с которым мы пока имели дело, называется *функциональным* (или *процедурным*) программированием и часто приводит к созданию так называемых монолитных приложений, все функции которых сконцентрированы в нескольких модулях кода (а то и вовсе в одном). В ООП обычно используется гораздо больше модулей, каждый из которых обеспечивает конкретные функции и может быть изолирован или даже полностью отделен от всех остальных. Такое модульное программирование обеспечивает гораздо большую гибкость и возможности для многократного использования кода.

Чтобы нагляднее показать, о чем идет речь, представьте, что высокопроизводительное компьютерное приложение является мощным гоночным автомобилем. При создании с помощью традиционных приемов программирования этот автомобиль будет представлять собой одно целое. Если понадобится что-то улучшить в этом автомобиле, его придется заменить целиком, т.е. отправить обратно изготовителю для переделки профессиональным механиком или вообще купить новый автомобиль. А технология ООП позволяет, например, купить у изготовителя новый двигатель и самостоятельно заменить его, следуя инструкциям изготовителя и не углубляясь в тонкости проведения ремонтных работ.

В традиционном приложении поток выполнения обычно прост и линеен. Приложения загружаются в память, начинают выполняться в точке А, завершают работу в точке Б и затем выгружаются из памяти. Попутно могут использоваться и другие разнообразные сущности, вроде файлов на носителе данных или возможностей видеокарты, но основная

часть обработки выполняется все-таки в одном месте. Сама обработка данных обычно несложная, использует различные математические и логические средства, а для построения более сложных представлений данных применяются простые типы вроде целочисленных или логических.

В ООП подобная линейность встречается редко. Результаты достигаются те же, но способ их получения зачастую выглядит совершенно по-другому. В ООП основной акцент делается на структуру и смысл данных, а также на взаимодействие этих данных с другими данными. Это обычно требует больше усилий на этапах проектирования приложения, но обеспечивает возможность его расширения. После принятия решения о представлении конкретных типов данных это представление может применяться в последующих версиях данного приложения и даже в совершенно новых приложениях. Наличие подобного соглашения может значительно сократить время разработки. Этим можно объяснить пример с гоночным автомобилем, где таким соглашением является структурирование кода "двигателя", который позволяет легко подставлять новый код (новый двигатель), не возвращая его изготовителю. Помимо этого, двигатель после создания можно использовать и для других целей — например, в другом автомобиле или вовсе в подводной лодке.

ООП часто упрощает программирование с помощью соглашений по представлению и применению более абстрактных объектов. Например, соглашение может приниматься не только по формату данных, используемых для отправки выходных данных на устройство вроде принтера, но и по методам обмена данными с этим устройством, т.е. инструкциям, которые оно должно понимать, и т.д. В автомобилестроении такое соглашение касалось бы способа подключения двигателя к топливной системе, трансмиссии и т.п.

Как видно из названия этой технологии, достигается это все с помощью *объектов*.

Что такое объект

Объект — это "строительный блок" в ООП-приложении. Такой строительный блок инкапсулирует часть приложения — процесс, порцию данных или какой-то более абстрактный объект.

Простейшие объекты могут быть очень похожи на знакомый нам тип структуры, который содержит члены типа переменных и функций. Содержащиеся переменные представляют хранимые в объекте данные, а содержащиеся функции обеспечивают доступ к возможным действиям этого объекта. Чуть более сложные объекты могут вообще не содержать данных, а представлять процесс и содержать только реализующие этот процесс функции. Примером такого объекта может служить принтер с функциями управления (распечатка документа, вывод тестовой страницы и т.д.).

Объекты в языке C# создаются из типов, как и хорошо знакомые составные переменные. Для типа объекта в ООП имеется специальное название — *класс*. Определения классов позволяют создавать объекты — т.е. реальные именованные *экземпляры* класса. Понятия "экземпляр класса" и "объект" эквивалентны, но термины "класс" и "объект" означают совершенно разные вещи.

Термины "класс" и "объект" часто путают, поэтому очень важно понимать, чем они отличаются. Помочь в этом может наша аналогия с гоночным автомобилем. Классом можно считать чертежи для изготовления автомобиля, а объектом — сам автомобиль, сделанный по этим чертежам.

В настоящей главе для работы с классами и объектами используется язык UML (Unified Modeling Language — унифицированный язык моделирования). Этот язык был специально разработан для моделирования программных систем — от объектов, из которых они состоят, и операций, которые они выполняют, до предполагаемых способов их использования. Здесь применяются и объясняются только базовые возможности языка UML, без более сложных аспектов, которым посвящены целые книги.

В VS имеется мощное средство для просмотра классов, которое позволяет отображать классы однотипным образом. Но для простоты все рисунки в данной главе выполнены вручную.

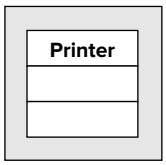


Рис. 8.1. *UML-представление класса* `Printer`

На рис. 8.1 для примера показано UML-представление класса принтера по имени `Printer`. Имя класса записано в самом верхнем разделе данного прямоугольника (о двух нижних разделах будет рассказано ниже).

На рис. 8.2 показано UML-представление экземпляра данного класса `Printer` по имени `myPrinter`.

Здесь в верхнем разделе записано имя экземпляра, а за ним, через двоеточие, имя его класса.

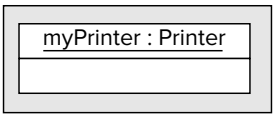


Рис. 8.2. *UML-представление экземпляра класса* `Printer` *по имени* `myPrinter`

Свойства и поля

Свойства и поля обеспечивают доступ к содержащимся в объекте данным. Эти данные как раз и являются тем, что отличает отдельные объекты друг от друга, поскольку в свойствах и полях разных объектов одного и того же класса могут храниться разные значения.

Различные фрагменты содержащихся в объекте данных вместе образуют *состояние* этого объекта. Например, пусть имеется класс объектов, представляющий чашку кофе и потому имеющий имя `CupOfCoffee`. При создании экземпляра (т.е. объекта) этого класса необходимо обеспечить его состояние, чтобы он был осмысленным. Для этого использующий данный объект код может с помощью свойств и полей задать сорт используемого кофе, содержится ли в кофе молоко и/или сахар, является ли кофе быстрорастворимым, и т.д. Тогда каждый конкретный объект `CupOfCofee` будет иметь определенное состояние, например: “Чашка колумбийского кофе с молоком и двумя кусочками сахара”.

Поля и свойства имеют типы, поэтому информация может в них храниться в виде значений `string`, `int` и т.д. Свойства отличаются от полей тем, что они не предоставляют непосредственный доступ к данным. Объекты могут ограждать пользователей от внутренних деталей своих данных, которые не обязательно взаимно однозначно представлены в существующих свойствах. Скажем, в поле для хранения информации о количестве кусочков сахара в экземпляре `CupOfCoffee` пользователи смогут помещать любые значения, ограниченные лишь пределами типа этого поля. То есть, например, в случае использования для хранения этих данных типа `int` пользователи смогут помещать в это поле любое значение в диапазоне от –2 147 483 648 до 2 147 483 647 (см. главу 3). Очевидно, что не все такие значения будут иметь смысл, особенно отрицательные, да и для слишком больших положительных может понадобиться чашка необычайно больших размеров. Использование свойства для хранения этой информации легко позволяет ограничить данное значение, скажем, только числами от 0 до 2.

В общем случае для доступа к состоянию лучше предоставлять свойства, а не поля, поскольку они позволяют управлять многими аспектами их поведения. Этот выбор никак не влияет на код, где применяются экземпляры объектов, т.к. синтаксис для использования свойств и полей выглядит одинаково.

Объект может четко определять и тип доступа к свойствам (для чтения и/или для записи). Некоторые свойства могут быть доступными только для чтения, т.е. они позволяют узнать их значения, но не изменять их (по крайней мере, непосредственно). Это часто бывает удобно для одновременного считывания нескольких фрагментов состояния. Например, класс `CupOfCoffee` может иметь доступное только для чтения свойство по имени `Description` (Описание), возвращающее строку с полной информацией о состоянии экземпляра этого класса. Конечно, эти сведения можно собрать и путем опроса нескольких свойств, но наличие такого свойства может сэкономить время и усилия. Аналогично ведут себя и свойства, доступные только для записи.

Кроме доступа для чтения и записи, для свойств и полей можно задавать разные виды *доступности*. Доступность членов класса определяет, какой код может получать доступ к этим членам, т.е. являются ли они доступными для всего кода (общедоступные), только для кода внутри класса (приватные) или следуют более сложной схеме (об этом будет рассказано позже). Как правило, поля делают приватными, а доступ к ним открывается через общедоступные свойства. При таком подходе код внутри класса имеет прямой доступ к хранящимся в поле данным, а общедоступное свойство ограждает внешних пользователей от этих данных и не позволяет им заносить туда недопустимое содержимое. Об общедоступных членах говорят, что они *предоставляются* классом.

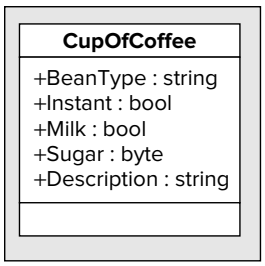


Рис. 8.3. *Отображение имен свойств и полей в UML-представлении класса*

Доступность немного похожа на область видимости переменных. Например, приватные поля и свойства можно считать локальными для объекта, который ими владеет, а область видимости общедоступных полей и свойств охватывает и внешний для объекта код.

В UML-представлении класса имена свойств и полей отображаются во втором разделе, как показано на рис. 8.3.

На этом рисунке показано UML-представление класса `CupOfCoffee`, в котором определены пять членов (свойства или поля, поскольку в UML между ними нет никакой разницы). В каждой строке содержится следующая информация.

- Доступность. Общедоступные члены помечены символом "+", а приватные — символом "–". Правда, обычно в настоящей главе приватные члены не будут приводиться на диаграммах, поскольку эта информация является внутренней для класса. Доступ для чтения или записи не обозначается никак.
- Имя члена.
- Тип члена.

Имена членов и их типы разделяются двоеточием.

Методы

Термином *метод* принято называть предоставляемую объектом функцию. Методы могут вызываться так же, как и любые другие функции, и так же возвращать значения и принимать параметры (функции были рассмотрены в главе 6).

Методы применяются для доступа к функциональным возможностям объекта. Подобно полям и свойствам, они могут быть общедоступными или приватными, ограничивая при необходимости доступ к ним из внешнего кода. Нередко методы используют в своих действиях состояние объекта и обращаются к приватным членам. Например, в классе `CupOfCoffee` может быть определен метод `AddSugar()`, предоставляющий более понятный синтаксис для увеличения количества сахара, чем действия с соответствующим свойством `Sugar`.

В UML-представлении методы классов отображаются в третьем разделе прямоугольников, как показано на рис. 8.4.

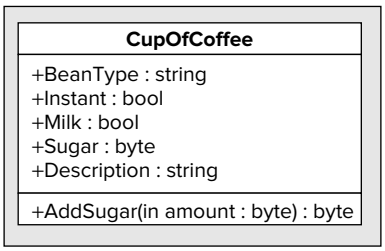


Рис. 8.4. *Отображение методов в UML-представлении класса*

Синтаксис записи методов в UML похож на синтаксис полей и свойств, только дополнительно записываются параметры методов, а тип в конце означает возвращаемый тип. Каждый параметр отображается в UML с одним из следующих идентификаторов: `in`, `out` или `inout`. Они обозначают направление потока данных, причем `out` и `inout` примерно соответствуют ключевым словам `out` и `ref` в языке C#, о которых рассказывалось в главе 6, а `in` — стандартному поведению в языке C#, когда не указаны ни `out`, ни `ref`.

Объектом является все, что угодно

Теперь пора внести ясность: объекты, свойства и методы уже не раз встречались в этой книге. Вообще-то в языке C# и .NET Framework объектом является все, что угодно. Функция `Main()` в консольном приложении является методом класса. Каждый из рассмотренных нами типов переменных является классом. Каждая из продемонстрированных команд, наподобие `<`*`строка`*`>.Length`, `<`*`строка`*`>.ToUpper()` и т.д. — это свойство или метод. (Символ точки здесь отделяет имя экземпляра объекта от имени свойства или метода; методы отличаются от свойств наличием скобок после них.)

Объекты встречаются повсюду, а синтаксис их использования обычно очень прост. До сих пор рассматривались лишь фундаментальные аспекты C#, и все приводимые примеры были достаточно простыми. С этого момента начнется более детальное изучение объектов. Очень важно запомнить, что описываемые здесь концепции имеют далеко идущие последствия — даже в отношении простой маленькой переменной `int`, с которой было так легко иметь дело.

Жизненный цикл объекта

Каждый объект имеет четко определенный жизненный цикл. Помимо обычного состояния — использование объекта — этот жизненный цикл включает два важных этапа.

- **Построение**. При создании экземпляра объекта его нужно инициализировать. Этот процесс инициализации и называется *построением*, и выполняется он функцией-конструктором, которую часто называют просто *конструктором*.
- **Уничтожение**. При уничтожении объекта часто требуется выполнить какие-либо операции по зачистке, вроде освобождения памяти. За их выполнение отвечает функция-деструктор, которую часто называют просто *деструктором*.

Конструкторы

Простая инициализация объекта осуществляется автоматически. Например, не нужно подыскивать место в памяти для размещения нового объекта. Однако иногда бывает необходимо выполнить на этапе инициализации объекта дополнительные задачи — например, инициализировать хранимые в объекте данные. Для этого применяется конструктор.

Все определения класса содержат, по крайней мере, один конструктор — *конструктор по умолчанию*, или *стандартный* конструктор — не принимающий параметров метод с таким же именем, как у самого класса. Определение класса может включать и несколько других конструкторов, принимающих параметры. Такие конструкторы позволяют создавать экземпляр объекта различными способами — например, предоставляя начальные значения для хранимых в объекте данных.

В языке C# конструкторы вызываются с помощью ключевого слова `new`. К примеру, создать экземпляр объекта `CupOfCoffee` с помощью конструктора по умолчанию можно следующим образом:

```
CupOfCoffee myCup = new CupOfCoffee();
```

Объекты можно создавать и с помощью конструкторов, отличных от конструктора по умолчанию. Например, у класса `CupOfCoffee` может существовать такой конструктор, принимающий параметр для указания типа кофейных зерен:

```
CupOfCoffee myCup = new CupOfCoffee("Blue Mountain");
```

Конструкторы, подобно полям, свойствам и методам, могут быть общедоступными или приватными. Код, внешний по отношению к классу, не может создавать объекты с помощью приватного конструктора; он должен обязательно использовать общедоступный конструктор. Так можно, например, заставить пользователей классов применять конструктор, отличный от конструктора по умолчанию (сделав конструктор по умолчанию приватным).

Некоторые классы не имеют общедоступных конструкторов, т.е. создавать их экземпляры из внешнего кода невозможно (такие классы называются *не создаваемыми*). Однако, как будет показано ниже, это не означает, что они совсем бесполезны.

Деструкторы

Деструкторы используются в .NET Framework для выполнения очистки при удалении объектов. Обычно какой-то особый код для деструктора не требуется; все, что нужно, делается автоматически. Однако можно добавить и специальные действия, если перед удалением объекта необходимо выполнять какие-либо важные операции.

Например, после выхода переменной из области видимости она может быть не доступной из кода, но по-прежнему существовать где-то в памяти. Только при выполнении средой .NET сборки мусора такой экземпляр уничтожается полностью.

Не надейтесь на освобождение деструктором ресурсов, которые используются экземпляром объекта: это может произойти спустя много времени после того, как объект станет ненужным. Если данные ресурсы являются критичными, это чревато появлением проблем. Однако существует одно решение, которое рассматривается ниже в разделе "Освобождаемые объекты".

Статические члены классов и члены экземпляров классов

Помимо свойств, методов и полей, принадлежащих конкретным экземплярам объектов, могут существовать и *статические* (еще называемые *разделяемыми*, особенно в Visual Basic) члены, которые тоже могут быть методами, свойствами и полями. Статические члены совместно используются разными экземплярами класса и поэтому могут считаться глобальными для объектов конкретного класса. Статические свойства и поля позволяют обращаться к данным, которые не зависят ни от каких экземпляров объектов, а статические методы — выполнять команды, связанные с типом класса, но не с конкретными экземплярами объектов. Более того, для использования статических членов даже не нужно создавать объекты.

Например, статическими являются методы `Console.WriteLine()` и `Convert.ToString()`, которые уже демонстрировались ранее в книге. Создавать экземпляры классов `Console` и `Convert` для них не нужно (да это и невозможно, т.к. конструкторы этих классов не являются общедоступными, как описывалось ранее).

Существует еще много подобных ситуаций, в которых статические свойства и методы можно применять с пользой. Например, статическое свойство можно использовать для отслеживания количества созданных экземпляров класса. В UML-синтаксисе статические члены классов обозначаются подчеркиванием, как показано на рис. 8.5.

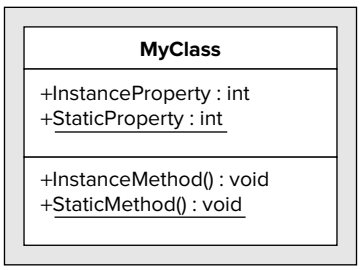


Рис. 8.5. *Статические члены в UML-представлении*

Статические конструкторы

Для применения статических членов класса может понадобиться сначала инициализировать их. Конечно, для статического члена можно указать начальное значение прямо в объявлении, но иногда нужна более сложная инициализация или, возможно, какие-то другие операции перед присваиванием значений или выполнением статических методов.

Для выполнения подобных задач инициализации предусмотрены статические конструкторы. У класса может быть только один статический конструктор, который не должен иметь модификаторов доступа и не может принимать никаких параметров. Статический конструктор вызывается не непосредственно, а в одном из следующих случаев:

- при создании экземпляра класса, содержащего данный статический конструктор;
- при обращении к статическому члену класса, содержащему данный статический конструктор.

В обоих случаях сначала вызывается статический конструктор, и только потом создается экземпляр класса или выполняется обращение к статическим членам. Сколько бы экземпляров класса ни создавалось, его статический конструктор вызывается только один раз. Чтобы отличать статические конструкторы от описанных ранее в этой главе, все не статические конструкторы называются также *конструкторами экземпляров*.

Статические классы

Часто бывает нужно использовать классы, содержащие только статические члены, для которых невозможно создавать объекты (вроде класса `Console`). Проще всего не делать все конструкторы класса приватными, а использовать *статический класс*. Статический класс может содержать только статические члены и по своей сути не может содержать конструкторы экземпляров. Однако в нем могут быть статические конструкторы, как было сказано в предыдущем разделе.

Новичкам в ООП не помешает сделать паузу перед изучением остального материала этой главы. Очень важно полностью разобраться со всеми базовыми понятиями и только затем приступать к рассмотрению более сложных аспектов.

Приемы объектно-ориентированного программирования

Теперь, когда изучены основы и известно, что такое объекты и как они работают, можно переходить к рассмотрению других функциональных возможностей объектов. В данном разделе речь пойдет о следующем:

- интерфейсы;
- наследование;
- полиморфизм;
- отношения между объектами;
- перегрузка операций;
- события;
- ссылочные типы и типы-значения.

Интерфейсы

Интерфейс (interface) — это коллекция общедоступных (а значит, не статических) методов и свойств, которые сгруппированы для инкапсуляции конкретной функциональности. После определения интерфейса его можно реализовать в классе. Это означает, что в таком случае класс будет поддерживать все свойства и члены, указанные данным интерфейсом.

Интерфейсы не существуют сами по себе. "Создать экземпляр" интерфейса, как для класса, нельзя. Кроме того, интерфейсы не могут содержать код с реализацией их членов, они могут лишь определять их. Реализация членов осуществляется в классах, реализующих данный интерфейс.

В приведенном ранее примере класса `CupOfCoffee` многие свойства и методы более общего назначения, вроде `AddSugar()`, `Milk`, `Sugar` и `Instant`, можно сгруппировать интерфейс с именем, скажем, `IHotDrink` (имена интерфейсов обычно начинаются с заглавной буквы `I`). Такой интерфейс можно применять для других объектов, например, объектов класса `CupOfTea`. Это позволило бы однотипно работать со всеми подобными объектами, хотя они могут иметь и собственные свойства (например, объекты `CupOfCoffee` — свойство `BeanType`, а объекты `CupOfTea` — свойство `LeafType`).

Интерфейсы, реализуемые для классов, изображаются в UML с помощью *кружочков*. На рис. 8.6 классы, реализующие интерфейс `IHotDrink`, вынесены в отдельный прямоугольник.

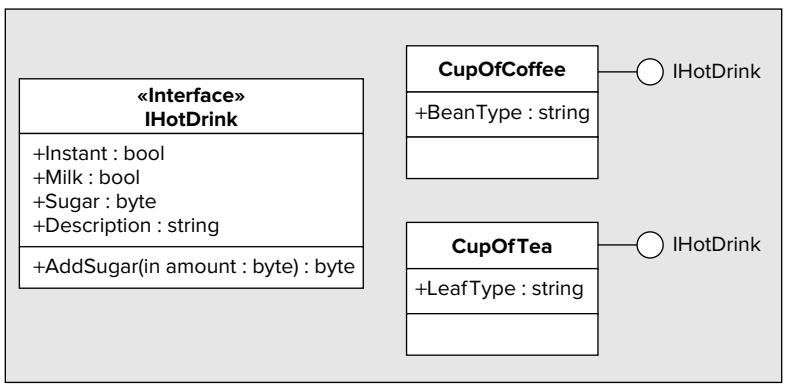


Рис. 8.6. *Представление интерфейсов в UML*

Один класс может поддерживать несколько интерфейсов, а несколько классов могут поддерживать один и тот же интерфейс. Значит, интерфейсы упрощают жизнь для пользователей и других разработчиков. Например, предположим, что имеется код, в котором используется объект с определенным интерфейсом. Если не использовать другие свойства и методы этого объекта, один объект можно будет легко заменять другим (код из примера с интерфейсом `IHotDrink`, например, может работать как с экземплярами `CupOfCoffee`, так и с экземплярами `CupOfTea`). Кроме того, разработчик данного класса сам может выпустить его обновленную версию, и если она поддерживает уже используемый интерфейс, другой разработчик сможет легко применить новую версию в своем коде.

После публикации интерфейса, т.е. предоставления к нему доступа другим разработчикам или конечным пользователям, лучше не изменять его. Интерфейс можно считать своего рода контрактом между создателями класса и его потребителями, в котором создатели заявляют, что каждый класс, реализующий интерфейс `X`, будет поддерживать такие-то методы и свойства. Если интерфейс позже изменится — например, из-за обновления базового кода — экземпляры классов будут работать неправильно или вообще не смогут работать. Поэтому вместо этого рекомендуется создавать новый интерфейс, расширяющий возможности старого и имеющий другой номер версии, вроде `X2`. Такой подход уже стал стандартным, поэтому интерфейсы с пронумерованными версиями встречаются довольно часто.

Освобождаемые объекты

Один из интерфейсов особенно интересен — это интерфейс `IDisposable`. Объект, поддерживающий этот интерфейс, должен реализовать метод `Dispose()`, т.е. предоставить код для этого метода. `Dispose()` может вызываться тогда, когда объект уже больше не нужен (например, непосредственно перед его выходом за пределы области видимости), и должен использоваться для освобождения любых критических ресурсов, которые в противном случае могут оставаться занятыми вплоть до вызова метода деструктора во время сборки мусора. Это обеспечивает больший контроль над ресурсами, которыми пользуются объекты.

В языке C# имеется конструкция специально для эффективного применения этого метода. Ключевое слово `using` позволяет инициализировать использующий критические ресурсы объект в кодовом блоке, при достижении конца которого автоматически вызывается метод `Dispose()`:

```
<имяКласса> <имяПеременной> = new <имяКласса>();
...
using (<имяПеременной>)
{
  ...
}
```

Объект *<имяПеременной>* можно создать и в составе оператора `using`:

```
using (<имяКласса> <имяПеременной> = new <имяКласса>)
{
  ...
}
```

И в том, и в другом случае переменная *<имяПеременной>* будет доступна внутри блока `using` и автоматически удалена в его конце (т.е. по завершении выполнения кода этого блока будет автоматически вызван метод `Dispose()`).

Наследование

Наследование (inheritance) — один из самых важных механизмов в ООП. Любой класс может наследоваться от другого класса, а это значит, что он будет иметь все те члены, что и класс, от которого он унаследован. В терминологии ООП класс, *от* которого наследуется

(порождается) другой класс, называется *родительским* или *базовым* классом. Классы в C# могут непосредственно наследоваться только от одного базового класса, хотя у того базового класса может быть собственный базовый класс, и т.д.

Механизм наследования позволяет расширять или создавать конкретные классы от одного более общего базового класса. Например, возьмем класс, представляющий животное с фермы. Этот класс мог бы называться `Animal` (Животное) и обладать методами вроде `EatFood()` (Питаться) или `Breed()` (Плодиться). От него можно создать производный класс по имени `Cow` (Корова), который поддерживает все эти методы, но при этом имеет и собственные — например, `Moo()` (Мычать) и `SupplyMilk()` (Давать молоко). Другим производным классом может быть класс `Chicken` (Курица) с методами `Cluck()` (Кудахтать) и `LayEgg()` (Снести яйцо).

В UML наследование изображается стрелками, как показано на рис. 8.7.

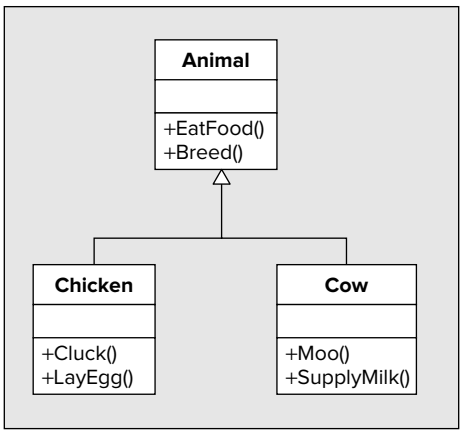


Рис. 8.7. *Представление отношений наследования в UML*

На рис. 8.7 возвращаемые типы членов для простоты не показаны.

При порождении от базового класса становится важным вопрос о доступности членов. Приватные члены базового класса недоступны из производного класса, а общедоступные (естественно) доступны. Но общедоступные члены доступны не только из производного класса, но и из внешнего кода. Поэтому если использовать только два этих уровня доступности, то невозможно создавать члены, доступные как из базового, так и из производного класса, но не из внешнего кода.

Для обхода этой проблемы существует третий уровень доступности — `protected` (защищенный), при котором доступ к члену имеют только производные классы. Для внешнего кода уровень доступности `protected` идентичен уровню `private`.

Помимо уровня защиты, для члена можно определить и способ наследования. Члены базового класса могут быть *виртуальными* (virtual), а это означает то, что они могут переопределяться в наследующем их классе, т.е. производный класс может содержать альтернативную реализацию таких членов. Эта альтернативная реализация не отменяет исходной реализации, которая все так же доступна в рамках исходного класса, но она закрывает ее от внешнего кода. При отсутствии альтернативной реализации любой внешний код, обращающийся к члену через производный класс, автоматически использует реализацию этого члена из базового класса.

Виртуальные члены не могут быть приватными: ведь невозможно, чтобы член мог переопределяться в производном классе и в то же время не был доступен из этого производного класса.

В примере с классом `Animal` можно сделать виртуальным метод `EatFood()` и предоставить для него новую реализацию в каком-то производном классе, например, в `Cow`, как показано на рис. 8.8. На этом рисунке метод `EatFood()` отображается и в классе `Animal`, и в классе `Cow`; это указывает на то, что у каждого из классов имеется собственная реализация данного метода.

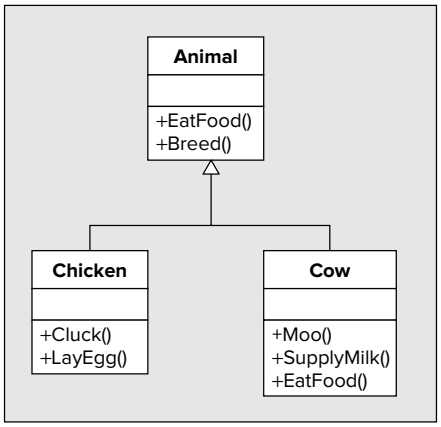


Рис. 8.8. *Представление виртуального метода в UML*

Базовые классы могут также определяться как *абстрактные* (abstract). Создавать экземпляр абстрактного класса непосредственно нельзя; использовать такой класс можно только для создания порожденных классов. У абстрактных классов могут быть абстрактные члены, которые не могут иметь реализацию в базовом классе — она должна быть представлена в производных классах. Например, если бы класс `Animal` был абстрактным, то в UML-представлении он выглядел бы так, как показано на рис. 8.9.

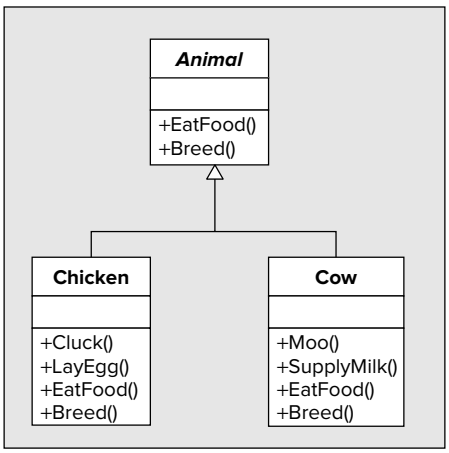


Рис. 8.9. *Представление абстрактных классов в UML*

Имена абстрактных классов записываются в UML курсивом (или выделяются пунктирной рамкой).

На рис. 8.9 методы `EatFood()` и `Breed()` указаны в абстрактных классах `Chicken` и `Cow`; это значит, что они являются либо абстрактными (и тогда подлежат переопределению в производных классах), либо виртуальными (т.е. уже определены в самих классах `Chicken` и `Cow`). Разумеется, абстрактные базовые классы могут содержать реализации для своих членов, и так нередко и бывает. Невозможность создания экземпляра абстрактного класса не означает, что в нем нельзя инкапсулировать функциональные возможности.

И, наконец, классы могут быть *запечатанными* (sealed). Такие классы не могут выступать в роли базового класса и, следовательно, не могут иметь производных классов.

В языке C# имеется один общий базовый класс для всех объектов, имеющий имя `object` (это псевдоним для класса `System.Object` из .NET Framework). Этот класс более подробно рассматривается в главе 9.

Интерфейсы, о которых рассказывалось в этой главе, тоже могут наследоваться от других интерфейсов. Но, в отличие от классов, они могут наследоваться от нескольких базовых интерфейсов (так же, как и классы могут поддерживать несколько интерфейсов).

Полиморфизм

Одним из результатов наследования является наличие в производных классах методов и свойств, совпадающих с базовым классом. Поэтому для экземпляров классов с общим базовым типом часто возможно применять идентичный синтаксис. Например, при наличии у класса `Animal` метода `EatFood()` синтаксис для вызова этого метода из производных классов `Cow` и `Chicken` будет одинаковым:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

Полиморфизм (polymorphism) позволяет двинуться еще дальше: присваивать значение переменной производного типа переменной базового типа:

```
Animal myAnimal = myCow;
```

Приведение при этом не требуется. Далее можно просто вызывать методы базового класса через эту переменную:

```
myAnimal.EatFood();
```

Данная строка кода приведет к вызову реализации метода `EatFood()` из производного класса. Однако вызывать подобным образом методы, определенные в производном классе, нельзя. То есть следующий код работать не будет:

```
myAnimal.Moo();
```

Хотя можно привести переменную типа базового класса к типу производного класса и вызвать метод производного класса:

```
Cow myNewCow = (Cow)myAnimal;
myNewCow.Moo();
```

Эта операция приведения приведет к исключению, если тип исходной переменной не совпадает ни с типом `Cow`, ни с типом производного от него класса. Для определения типа объекта существуют свои приемы, которые будут описаны в следующей главе.

Полиморфизм чрезвычайно полезен тем, что минимизирует объем кода для выполнения действий с разными объектами, происходящими от одного класса. Полиморфизм может применяться не только в отношении классов, имеющих одинаковый родительский класс. Он может применяться и, скажем, в отношении дочерних и "внучатых" классов — главное, чтобы в их иерархии наследования имелся общий класс.

Еще раз напоминаем, что в C# все классы порождаются от базового класса `object`, который является корневым в их иерархиях наследования. Поэтому все объекты можно считать экземплярами класса `object`. Это и позволяет методу `Console.WriteLine()` при построении строк обрабатывать практически бесконечное количество комбинаций параметров. Каждый параметр после первого считается экземпляром `object`, что позволяет выводить данные любого объекта. Для этого вызывается метод `ToString()`, являющийся членом класса `object`. Этот метод можно переопределить более подходящей реализацией, а можно просто воспользоваться стандартной реализацией, в которой он возвращает имя класса (уточненное пространствами имен, в которых он присутствует).

Полиморфизм интерфейсов

Создавать экземпляры интерфейсов таким же образом, как и экземпляры объектов, нельзя, но можно создать переменную типа интерфейса и затем использовать ее для обращения к методам и свойствам, предоставляемым этим интерфейсом в объектах, которые его поддерживают.

Например, предположим, что вместо базового класса `Animal` метод `EatFood()` помещен в интерфейс по имени `IConsume`. Этот интерфейс могут поддерживать оба класса — `Cow` и `Chicken`, — только каждый из них должен иметь свою реализацию метода `EatFood()`, поскольку интерфейсы не содержат реализаций. После этого к данному методу можно обращаться с помощью примерно такого кода:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface = myCow;
consumeInterface.EatFood();
consumeInterface = myChicken;
consumeInterface.EatFood();
```

Получается простой способ для однотипного вызова различных методов, который не зависит от общего базового класса. Например, такой интерфейс можно реализовать и в классе `VenusFlyTrap` (Мухоловка), порожденном не от класса `Animal` (Животное), а от класса `Vegetable` (Растение):

```
VenusFlyTrapmyVenusFlyTrap=newVenusFlyTrap();
IConsumeconsumeInterface;
consumeInterface=myVenusFlyTrap;
consumeInterface.EatFood();
```

В этом коде вызов `consumeInterface.EatFood()` приводит к вызову метода `EatFood()` класса `Cow`, или `Chicken`, или `VenusFlyTrap` — в зависимости от того, какой экземпляр будет присвоен переменной типа интерфейса.

Производные классы наследуют интерфейсы, поддерживаемые их базовыми классами. В первом из предыдущих примеров интерфейс `IConsume` может поддерживаться как классом `Animal`, так и обоими классами `Cow` и `Chicken`. Учтите, что классы с общим базовым классом не обязательно имеют общие интерфейсы, и наоборот.

Отношения между объектами

Наследование является простым отношением между объектами, когда базовый класс полностью отражен производным классом, и производный класс может также иметь дос-

туп к внутренним деталям своего базового класса (через защищенные члены). Однако бывают и другие ситуации, в которых отношения между объектами более важны.

В настоящем разделе будут кратко рассмотрены следующие отношения.

- **Включение**. Один класс содержит другой. Такие отношения похожи на наследование, но позволяют содержащему классу управлять доступом к членам содержащегося внутри него класса и даже выполнять дополнительную обработку перед использованием этих членов.
- **Коллекции**. Один класс выступает в роли контейнера для нескольких экземпляров другого класса. Это похоже на массивы объектов, но у коллекций имеются и другие возможности: индексация, поиск, изменение размера и т.д.

Включение

Включение легко достигается использованием поля-члена для хранения экземпляра объекта. Это поле может быть общедоступным, и тогда пользователи содержащего объекта будут иметь доступ к предоставляемым им методам и свойствам — примерно как при наследовании. Но доступа к внутренним деталям класса через производный класс, который возможен в случае наследования, все-таки не будет.

Содержащийся внутри объект-член можно сделать приватным членом. В таком случае ни один из его членов не будет доступен пользователям непосредственно, даже если они являются общедоступными. Тогда обращаться к этим членам можно с помощью членов класса-контейнера. Такой подход позволяет полностью управлять тем, какие члены содержащегося внутри класса должны предоставляться (и должны ли вообще), а также выполнять дополнительную обработку в членах класса-контейнера перед обращением к членам содержащегося класса.

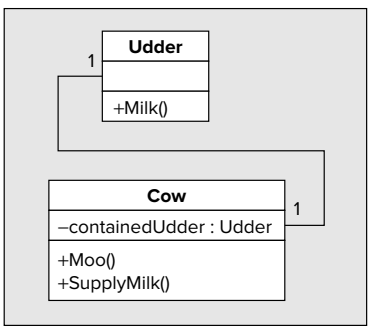


Рис. 8.10. *Представление включения в UML*

Например, класс `Cow` может содержать класс `Udder` (Вымя) с общедоступным методом `Milk()` (Доить). Объект `Cow` может вызывать этот метод — например, в составе своего метода `SupplyMilk()`, но пользователям объекта `Cow()` такие детали не видны (и не важны).

Содержащиеся внутри других классы в UML могут изображаться с помощью связующих линий. В случае простого включения концы этих линий обозначаются единицами (1), что означает тип отношения один к одному (т.е., например, что один экземпляр `Cow` будет содержать один экземпляр `Udder`). Для большей наглядности экземпляр класса `Udder` может быть также изображен в виде приватного поля класса `Cow`, как показано на рис. 8.10.

Коллекции

В главе 5 были описаны массивы, предназначенные для хранения нескольких переменных одинакового типа. То же самое можно делать и для объектов (и неудивительно: вспомните, что уже знакомые вам типы переменных на самом деле являются объектами). Например:

```
Animal[] animals = new Animal[5];
```

Коллекция — это, по сути, массив, но с дополнительными возможностями. Коллекции реализуются в виде классов практически так же, как и другие объекты. Их имена часто представляют собой множественное число имени хранимых в них объектов: например, класс с именем `Animals` может содержать коллекцию объектов `Animal`.

Главное отличие от массивов состоит в том, что коллекции обычно реализуют дополнительные функции, вроде методов `Add()` и `Remove()` для добавления элементов в коллекцию и удаления из нее. У них также обычно имеется свойство `Item`, которое возвращает объект по его индексу. Довольно часто это свойство реализуется так, чтобы был возможен более сложный доступ. Например, класс `Animals` можно спроектировать так, чтобы обращаться к каждому конкретному объекту `Animal` по его имени.

В UML подобные отношения изображаются так, как показано на рис. 8.11.

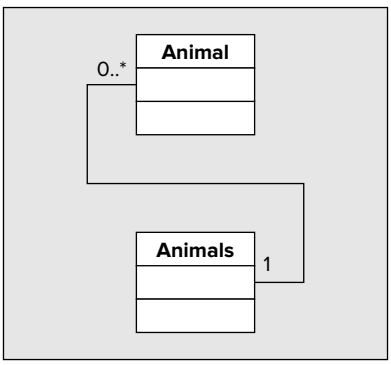


Рис. 8.11. *Представление коллекции в UML*

Члены на этом рисунке не показаны, т.к. здесь рассматриваются отношения. Числа на концах связующих линий указывают, что один объект `Animals` содержит ноль или более объектов `Animal`. Коллекции будут более подробно рассмотрены в главе 11.

Перегрузка операций

Вы уже знаете, как применять операции для обработки переменных простых типов. В некоторых случаях естественно использовать операции и с объектами собственных классов. Это возможно, потому что классы могут содержать инструкции по выполнению операций.

Например, в класс `Animal` можно добавить новое свойство `Weight` (Вес); тогда можно сравнивать вес животных с помощью такого кода:

```
if (cowA.Weight > cowB.Weight)
{
  ...
}
```

Перегрузка операций позволяет предоставить логику неявного использования свойства `Weight`, чтобы можно было написать так:

```
if (cowA > cowB)
{
  ...
}
```

Здесь операция "больше" (>) *перегружена*. Перегруженной называется такая операция, для которой был написан выполняющий ее код; этот код добавляется в определение одного из классов, для которого она должна выполняться. В предыдущем примере используются два объекта `Cow`, поэтому определение перегрузки операции содержится в классе `Cow`. Аналогично можно перегружать операции и для работы с разными классами — тогда соответствующий код должен находиться в одном из определений классов (или в обоих).

Перегрузка возможна только для операций, существующих в языке C#: новые операции создавать нельзя. Однако реализации можно предоставлять как для унарных, так и для бинарных версий операции вроде +. Как это сделать, будет рассказано в главе 13.

События

Объекты при их работе могут генерировать (и обрабатывать) *события* (event). События важны тем, что позволяют выполнять определенные действия в других частях кода — этим они похожи на исключения, но мощнее их. Например, при добавлении объекта `Animal` в коллекцию `Animals` может понадобиться выполнять определенный код, не являющийся ни частью класса `Animals`, ни частью того кода, который вызывает метод `Add()`. Для этого потребуется добавить в код *обработчик события* — особую функцию, которая вызывается при возникновении события. Кроме того, нужно настроить этот обработчик, чтобы он ожидал возникновения именно интересующего события.

С помощью событий можно создавать *управляемые событиями* приложения, которые гораздо более полезны, чем может показаться поначалу. Достаточно вспомнить, что, например, все Windows-приложения полностью зависят от событий. Каждый щелчок пользователя на кнопке или перетаскивание ползунка на полосе прокрутки выполняется с помощью обработки событий, генерируемых мышью или клавиатурой.

Как именно это все происходит в Windows-приложениях, будет показано ниже в этой главе, а вообще события детально рассматриваются в главе 13.

Ссылочные типы и типы значения

Данные в C# сохраняются в переменной одним из двух способов, который зависит от типа переменной. Этот тип относится к одной из двух категорий: ссылка или значение. Основные их отличия описаны ниже.

- Типы значения хранят себя и свое содержимое в одном месте в памяти.
- Ссылочные типы хранят ссылку на другое место в памяти (называемое *кучей* (heap)), в котором и хранится их содержимое.

Вообще-то при работе с C# особо беспокоиться об этом не нужно. Пока что в этой книге переменные типа `string` (ссылочный тип) и другие простые переменные (большинство из которых относится к типам значения, как, например, `int`) применялись практически одинаково.

Одно из главных отличий между типами значения и ссылочными типами состоит в том, что типы значения всегда содержат значения, а ссылочные типы могут быть пустыми (`null`), т.е. вообще не содержать значения. Хотя можно создать тип-значение, ведущий себя в этом отношении подобно ссылочному типу (может быть нулевым), путем использования *типов, допускающих нулевые значения* (nullable), которые являются разновидностью *обобщений* (generics). Обобщения представляют собой усовершенствованную технологию и более подробно рассматриваются в главе 12.

Единственными простыми ссылочными типами являются `string` и `object`, хотя неявно к ним относятся и массивы. Каждый создаваемый класс представляет собой ссылочный тип, поэтому здесь речь пойдет именно о них.

Главное отличие между структурами и классами состоит в том, что структуры представляют собой типы значения. Вы, наверно, обращали внимание на сходство структур и классов, особенно в главе 6, где было продемонстрировано использование функций в структурах. Более подробно об этом пойдет речь в главе 9.

Объектно-ориентированное программирование в Windows-приложениях

В главе 2 был приведен пример создания простого Windows-приложения на языке C#. Windows-приложения очень сильно зависят от ООП, и поэтому в настоящем разделе будут продемонстрированы некоторые его аспекты. В следующем практическом занятии предлагается еще один простой пример.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Объекты в действии

1. Создайте новое Windows-приложение с именем `Ch08Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter08`.
2. Добавьте в него новый элемент управления `Button` с панели Toolbox и поместите его в центр формы `Form1`, как показано на рис. 8.12.
3. Дважды щелкните на элементе `Button`, чтобы добавить код для обработки щелчка. Измените автоматически сгенерированный код следующим образом:

```
private void button1_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!";       // Выполнен щелчок
   Button newButton = new Button();
   newButton.Text = "New Button!";           // Новая кнопка!
   newButton.Click += new EventHandler(newButton_Click);
   Controls.Add(newButton);
}
private void newButton_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!!";
}
```

Фрагмент кода ***Ch08Ex01\Form1.cs***

4. Запустите приложение. После этого на экране должна появиться форма, выглядящая так, как показано на рис. 8.13.
5. Щелкните на кнопке с надписью button1. Изображение на экране должно измениться (рис. 8.14).

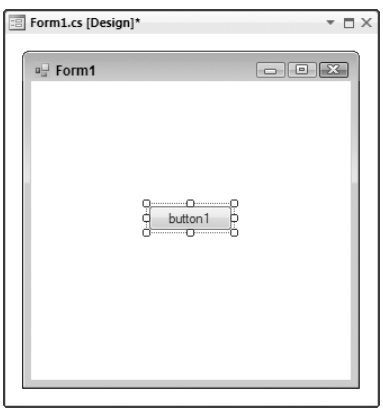


Рис. 8.12. *Окно с кнопкой*

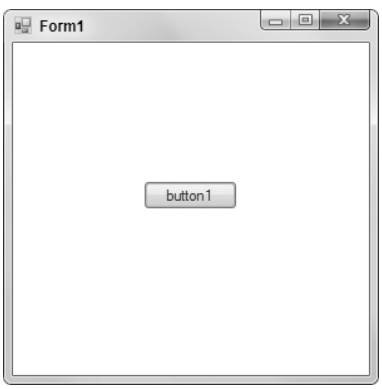


Рис. 8.13. *Приложения* `Ch08Ex01` *сразу после запуска*

6. Щелкните на кнопке с надписью New Button!. После этого изображение на экране должно измениться так, как показано на рис. 8.15.

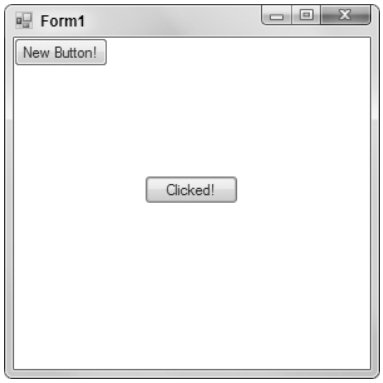


Рис. 8.14. *Приложение* `Ch08Ex01` *после щелчка на кнопке* button1

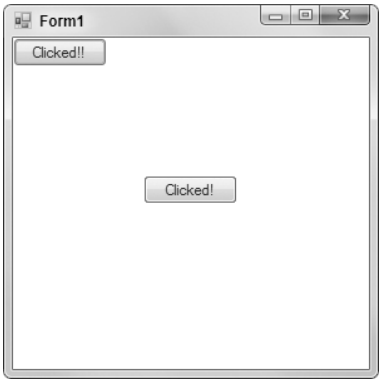


Рис. 8.15. *Приложение* `Ch08Ex01` *после щелчка на кнопке* New Button!

Описание работы

Добавление лишь нескольких строк кода позволило создать Windows-приложение, которое кое-что делает и демонстрирует применение некоторых приемов ООП в C#. Выражение "объектом является все, что угодно" в случае Windows-приложений справедливо еще более. Начиная с выполняемой формы и заканчивая элементами управления на этой форме — везде нужны приемы ООП. В данном упражнении были задействованы несколько рассмотренных в этой главе понятий, чтобы показать, как они работают все вместе.

Сначала на форму `Form1` приложения была добавлена новая кнопка. Кнопка представляет собой объект `Button`, а форма — объект `Form1`, порожденный от класса `Form`. Далее после двойного щелчка на кнопке был добавлен обработчик события для перехвата события `Click`, генерируемого объектом `Button`. Этот обработчик добавлен в код объекта `Form`, инкапсулирующего приложение, в виде приватного метода:

```
private void button1_Click(object sender, System.EventArgs e)
{
}
```

В этом коде используется спецификатор — ключевое слово `private`. Пока не обращайте на это внимания; код C#, необходимый для применения приемов ООП, описанных в настоящей главе, будет разъяснен в следующей главе.

В первой добавленной строке кода изменяется текст на кнопке, на которой выполняется щелчок, здесь задействуется полиморфизм, о котором рассказывалось ранее в главе. Объект `Button`, представляющий эту кнопку, отправляется обработчику событий через параметр `object`, который приводится к типу `Button` (это возможно, поскольку класс `Button` порожден от `System.Object`, т.е. класса .NET с псевдонимом `object`). Затем изменяется свойство `Text` этого объекта для изменения отображаемого текста:

```
((Button)sender).Text = "Clicked!";
```

Далее с помощью ключевого слова `new` создается новый объект `Button` (в этом проекте заданы пространства имен, позволяющие применять простой синтаксис; иначе для создания этого объекта пришлось бы использовать полностью квалифицированное имя `System.Windows.Forms.Button`):

```
Button newButton = new Button();
newButton.Text = "New Button!";
```

Ниже в коде добавлен соответствующий новый обработчик событий, который используется для реагирования на событие `Click`, генерируемое новой кнопкой:

```
private void newButton_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!!";
}
```

Затем этот обработчик регистрируется в качестве слушателя события `Click` с использованием синтаксиса перегрузки операций. С помощью конструктора создается новый объект `EventHandler` с именем новой функции-обработчика событий:

```
newButton.Click += new EventHandler(newButton_Click);
```

В конце используется свойство `Controls`. Это объект, представляющий коллекцию всех имеющихся на форме элементов управления, а его метод `Add()` применяется для добавления на форму новой кнопки:

```
Controls.Add(newButton);
```

Свойство `Controls` демонстрирует, что свойства не обязательно должны иметь простые типы вроде строк или целых чисел: они могут представлять собой любые объекты. В этом коротком примере были использованы почти все из описанных в настоящей главе приемов. Как видите, объектно-ориентированное программирование не обязательно должно быть сложным — это просто другой взгляд на программирование.

Резюме

В этой главе было приведено полное описание приемов объектно-ориентированного программирования — в контексте программирования на C#, но на принципы это не влияет. То есть большая часть материала данной главы описывает ООП на любом языке.

Сначала были рассмотрены основные вопросы: что означает термин “объект”, и как объект может быть экземпляром класса. Далее было рассказано, что объекты могут содержать разные члены — поля, свойства и методы, что эти члены могут иметь ограниченную доступность, и чем общедоступные члены отличаются от приватных. Потом вы узнали, что члены могут быть еще и защищенными, а также виртуальными и абстрактными (и что абстрактные методы возможны только в абстрактных классах). Вы также узнали, чем статические (совместно используемые) члены отличаются от членов экземпляров и почему иногда лучше использовать статические классы.

Затем был вкратце рассмотрен жизненный цикл объектов, наряду с их созданием конструкторами и удалением с помощью деструкторов. Позже, после рассмотрения группировки членов в интерфейсах, была описана и более сложная методика уничтожения объектов с помощью освобождаемых объектов, поддерживающих интерфейс `IDisposable`.

Остальная часть главы была в основном посвящена перечислению возможностей ООП, многие из которых будут более подробно рассмотрены в последующих главах. Вы познакомились с наследованием, позволяющим порождать классы от базовых классов, с двумя разновидностями полиморфизма (с помощью базовых классов и совместно используемые интерфейсов) и с включением в объект одного или более других объектов (с помощью включения и коллекций). И в конце было описано упрощение синтаксиса использования объектов с помощью перегрузки операций и то, как объекты генерируют события.

В заключительной части главы большая часть рассмотренных теоретических сведений была продемонстрирована на примере создания простого Windows-приложения. В следующей главе речь пойдет об определении классов в C#.

Упражнения

1. Какие из перечисленных ниже уровней доступности действительно существуют в ООП?

   а) `friend`

   б) `public`

   в) `secure`

   г) `private`

   д) `protected`

   е) `loose`

   ж) `wildcard`

2. "Деструктор объекта нужно обязательно вызывать вручную, иначе память будет использоваться неэффективно". Верно или нет?

3. Нужно ли создавать объект для вызова статического метода его класса?

4. Нарисуйте UML-диаграмму, подобную приведенным в главе, для перечисленных ниже классов и интерфейса.

   - Абстрактный класс с именем `HotDrink` (Горячий напиток), методами `Drink` (Пить), `AddMilk` (Добавить молока) и `AddSugar` (Добавить сахара) и свойствами `Milk` (Молоко) и `Sugar` (Сахар).
   - Интерфейс `ICup` (Чашка), содержащий методы `Refill` (Наполнить снова) и `Wash` (Помыть), а также свойства `Color` (Цвет) и `Volume` (Объем).
   - Класс с именем `CupOfCoffee` (Чашка кофе), порожденный от класса `HotDrink` (Горячий напиток), поддерживающий интерфейс `ICup` и обладающий дополнительным свойством `BeanType` (Сорт кофе).
   - Класс с именем `CupOfTea` (Чашка чая), порожденный от класса `HotDrink`, поддерживающий интерфейс `ICup` и обладающий дополнительным свойством `LeafType` (Сорт чая).

5. Напишите код для функции, которая в качестве параметра принимает один из двух возможных в предыдущем примере объектов типа `Cup...` (`CupOfCoffee` или `CupOfTea`). Эта функция должна вызывать методы `AddMilk`, `Drink` и `Wash` для любого передаваемого ей объекта `Cup`.

   Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Объекты и классы	Объекты — строительные блоки приложений на основе ООП. Классы — это определения типов, которые используются для создания объектов. Объекты могут содержать данные и/или предоставлять операции для выполнения в других частях кода. Данные можно сделать доступными для внешнего кода с помощью свойств, а операции — с помощью методов. Свойства и методы вместе называются членами класса. Свойства могут иметь разрешение на чтение, запись или и то, и другое. Члены класса могут быть общедоступными (доступными для всего кода) или приватными (доступными только в коде определения класса). В .NET все является объектом.
Жизненный цикл объекта	Объект создается с помощью вызова одного из конструкторов класса. Когда объект становится уже не нужным, он уничтожается с помощью деструктора. Для зачистки после использования объекта часто бывает необходимо избавляться от него вручную.
Статические члены и члены экземпляров	Члены экземпляров доступны только в объектах — экземплярах класса. Статические члены доступны только непосредственно через определение класса, они не связаны ни с каким экземпляром класса.
Интерфейсы	Интерфейс — это коллекция общедоступных свойств и методов, которые могут быть реализованы в классе. Переменной с типом экземпляра класса можно присвоить значение любого объекта, определение класса которого реализует этот интерфейс. Тогда через эту переменную будут доступны члены, определенную в данном интерфейсе.
Наследование	Наследование — это механизм порождения определения одного класса на основе другого. Дочерний класс наследует члены от (единственного) родительского класса. Дочерние классы не могут наследовать приватные члены своего родителя, но можно определить защищенные члены, которые доступны как внутри самого класса, так и внутри классов, порожденных от данного класса. Дочерние классы могут перекрывать члены, определенные в родительском классе как виртуальные. Цепочки наследования всех классов заканчиваются на `System.Object`, который в C# имеет псевдоним `object`.
Полиморфизм	Все объекты, созданные на основе порожденного класса, можно считать экземплярами родительского класса.
Отношения между объектами и дополнительные возможности	Объекты могут содержать другие объекты, а также могут представлять собой коллекции других объектов. Для работы с объектами в выражениях можно определить способ их обработки операциями с помощью перегрузки операций. Объекты могут предоставлять события, которые генерируются в каких-то внутренних процессах, а клиентский код может реагировать на события с помощью обработчиков событий.

9

Определение классов

В ЭТОЙ ГЛАВЕ...

- Определение классов и интерфейсов в C#
- Применение ключевых слов, управляющих доступностью и наследованием
- Класс `System.Object` и его роль в определениях классов
- Полезные средства в VS и VCE
- Определение библиотек классов
- Сходство и различие интерфейсов и абстрактных классов
- Дополнительные сведения о структурах
- Важная информация о копировании объектов

В главе 8 были рассмотрены теоретические аспекты объектно-ориентированного программирования (ООП). В этой главе теория будет воплощена в практику на примере определения классов в C#. Способы определения членов классов здесь не рассматриваются, потому что мы будем изучать определение самих классов. Может показаться, что это слишком узкая тема, но не волнуйтесь — материла для изучения более чем достаточно.

Сначала мы рассмотрим простой синтаксис для определения базовых классов, ключевые слова для указания доступности класса и не только, а также способ указания наследования. Кроме того, здесь будут рассмотрены определения интерфейсов, поскольку они во многом похожи на определения классов.

В остальной части будут освещены другие вопросы, связанные с определением классов в языке C#.

Определение классов в C#

В языке C# для определения классов применяется ключевое слово `class`:

```
class MyClass
{
   // Члены класса.
}
```

В этом коде определяется класс по имени `MyClass`. После определения класса его экземпляр можно создавать в любом месте проекта, где имеется доступ к этому определению. По умолчанию классы объявляются как *внутренние*, что позволяет обращаться к ним только из кода текущего проекта. Это можно (хотя и не обязательно) указать явно с помощью ключевого слова `internal`:

```
internal class MyClass
{
   // Члены класса.
}
```

Но можно указать, что класс является общедоступным и должен быть доступен в коде других проектов. Для этого предназначено ключевое слово `public`:

```
public class MyClass
{
   // Члены класса.
}
```

Классы, объявляемые так, как здесь, сами по себе не могут быть ни приватными, ни защищенными. В следующей главе будет показано, как с помощью соответствующих модификаторов объявить приватными или защищенными классы, являющиеся членами других классов.

Помимо этих двух модификаторов, можно указать, что класс является *абстрактным* (невозможно создавать объекты, может лишь порождать другие классы и иметь абстрактные члены), либо *запечатанным* (невозможно порождение производных классов). Для этого нужно указать одно из двух взаимно исключающих ключевых слов: `abstract` или `sealed`. Абстрактный класс объявляется следующим образом:

```
public abstract class MyClass
{
   // Члены класса, могут быть абстрактными.
}
```

Здесь `MyClass` представляет собой общедоступный абстрактный класс, в котором возможны внутренние абстрактные классы.

Запечатанные классы объявляются так:

```
public sealed class MyClass
{
   // Члены класса.
}
```

Подобно абстрактным, запечатанные классы могут быть как общедоступными, так и внутренними.

В определении класса можно указать наследование. Для этого после имени класса добавляется через двоеточие имя базового класса:

```
public class MyClass : MyBase
{
   // Члены класса.
}
```

В определениях классов C# разрешен только один базовый класс, а при наследовании от абстрактного класса необходимо реализовать все наследуемые абстрактные члены (если только производный класс сам не является абстрактным).

Компилятор не допускает, чтобы производный класс был более доступным, чем его базовый класс. Это означает, что внутренний класс можно породить от общедоступного класса, а общедоступный класс от внутреннего базового — нет. Следующий код скомпилируется нормально:

```
public class MyBase
{
   // Члены класса.
}
internal class MyClass : MyBase
{
   // Члены класса.
}
```

А этот код скомпилировать не удастся:

```
internal class MyBase
{
   // Члены класса.
}
public class MyClass : MyBase
{
   // Члены класса.
}
```

Если базовый класс не указан, класс наследуется от базового класса `System.Object` (который в C# имеет псевдоним `object`). В конечном счете, класс `System.Object` является корневым в иерархии наследования всех классов. Мы еще поговорим несколько позже об этом фундаментальном классе.

Подобным образом после двоеточия можно указывать не только базовые классы, но и поддерживаемые интерфейсы. Если задается базовый класс, он должен указываться после двоеточия первым, а уже за ним — интерфейсы. Если же базовый класс не задан, то интерфейсы указываются сразу после двоеточия. Для разделения имени базового класса (если указано) и интерфейсов служат запятые.

Например, в определение класса `MyClass` можно добавить интерфейс следующим образом:

```
public class MyClass : IMyInterface
{
   // Члены класса.
}
```

Все члены интерфейса обязательно должны быть реализованы в каком-то поддерживающем этот интерфейс классе, хотя если с данным членом интерфейса не требуется ничего делать, можно записать "пустую" реализацию (без функционального кода), а также реализовать члены интерфейса в виде абстрактных членов абстрактных классов.

Следующее объявление недопустимо, т.к. базовый класс `MyBase` не является первым в списке наследования:

```
public class MyClass : IMyInterface, MyBase
{
   // Члены класса.
}
```

Правильный способ указания базового класса и интерфейса выглядит следующим образом:

```
public class MyClass : MyBase, IMyInterface
{
   // Члены класса.
}
```

Поскольку интерфейсов может быть несколько, следующий код также допустим:

```
public class MyClass : MyBase, IMyInterface, IMySecondInterface
{
   // Члены класса.
}
```

В табл. 9.1 перечислены все сочетания модификаторов доступа, допустимые в определениях классов.

Таблица 9.1. Допустимые сочетания модификаторов доступа для определений классов

Модификатор	Описание
Отсутствует или `internal`	Класс, доступ к которому возможен только из текущего проекта
`public`	Класс, доступ к которому возможен из любого места
`abstract` или `internal abstract`	Класс, доступ к которому возможен только из текущего проекта и который не допускает создания экземпляров, т.е. позволяет создавать только производные классы
`public abstract`	Класс, доступ к которому возможен из любого места и который не допускает создания экземпляров, т.е. позволяет создавать только производные классы
`sealed` или `internal sealed`	Класс, доступ к которому возможен только из текущего проекта и который не допускает создания производных классов, т.е. допускает создание только экземпляров
`public sealed`	Класс, доступ к которому возможен из любого места и который не допускает создания производных классов, т.е. допускает создание только экземпляров

Определения интерфейсов

Интерфейсы объявляются примерно как и классы, но только с использованием ключевого слова `interface`, а не `class`:

```
interface IMyInterface
{
   // Члены интерфейса.
}
```

Модификаторы public и internal применяются точно так же, и подобно классам, интерфейсы по умолчанию объявляются внутренними. Чтобы сделать интерфейс общедоступным, используется ключевое слово public:

```
public interface IMyInterface
{
   // Члены интерфейса.
}
```

Ключевые слова abstract и sealed недопустимы, поскольку не имеют смысла в контексте интерфейсов (они не содержат реализации, поэтому для них невозможно непосредственно создавать экземпляры, а чтобы быть полезными, они должны допускать наследование).

Наследование интерфейсов также похоже на классы. Главное отличие состоит в том, что в случае интерфейсов можно указать несколько базовых интерфейсов:

```
public interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
{
   // Члены интерфейса.
}
```

Интерфейсы не являются классами и поэтому не наследуются от System.Object. Однако — исключительно для удобства — члены System.Object доступны через переменные типа интерфейса. Кроме того, как уже было сказано, нельзя создать экземпляр интерфейса так же, как экземпляр класса. В следующем практическом занятии приводится пример некоторых определений классов вместе с использующим их кодом.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Определение классов

1. Создайте новое консольное приложение с именем Ch09Ex01 и сохраните его в каталоге C:\BegVCSharp\Chapter09.

2. Измените код в Program.cs следующим образом:

```
namespace Ch09Ex01
{
   public abstract class MyBase
   {
   }

   internal class MyClass : MyBase
   {
   }

   public interface IMyBaseInterface
   {
   }

   internal interface IMyBaseInterface2
   {
   }

   internal interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
   {
   }

   internal sealed class MyComplexClass : MyClass, IMyInterface
   {
   }
```

```
    class Program
    {
       static void Main(string[] args)
       {
          MyComplexClass myObj = new MyComplexClass();
          Console.WriteLine(myObj.ToString());
          Console.ReadKey();
       }
    }
}
```

Фрагмент кода **Ch09Ex01\Program.cs**

3. Запустите приложение. На рис. 9.1 показан результат, который должен получиться.

Рис. 9.1. *Приложение* `Ch09Ex01` *в действии*

Описание работы

Иерархия наследования классов и интерфейсов, определенных в этом проекте, показана на рис. 9.2.

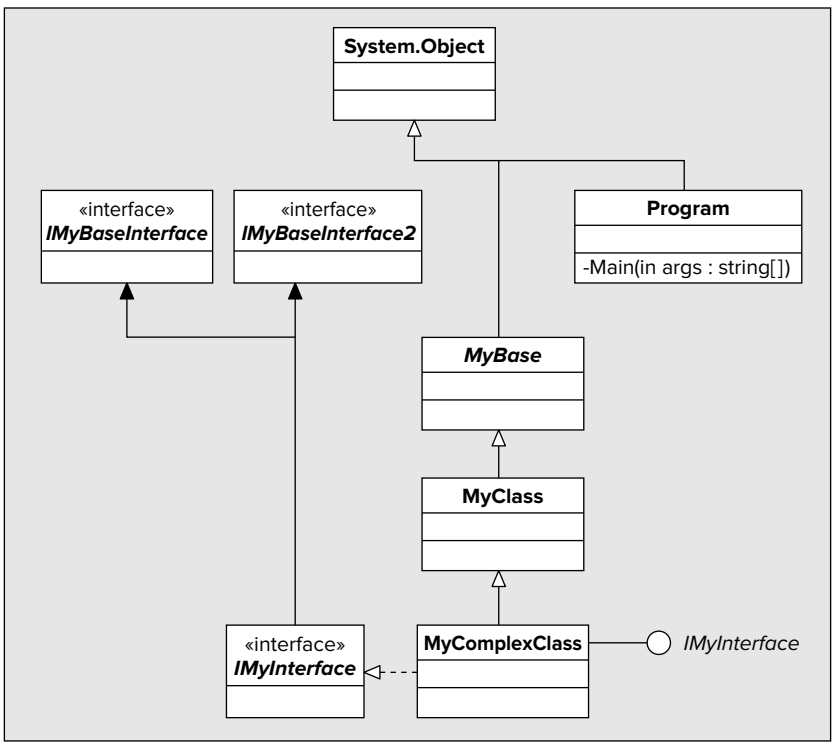


Рис. 9.2. *Иерархия наследования классов и интерфейсов в проекте* `Ch09Ex01`

Здесь включен и класс Program, потому что он определяется подобно другим классам, хотя и не входит в основную иерархию. Метод Main() этого класса является точкой входа приложения.

Определения MyBase и IMyBaseInterface являются общедоступными, т.е. эти классы доступны и в других проектах. Остальные классы и интерфейсы являются внутренними, и, следовательно, доступны только в данном проекте.

Код в Main() вызывает метод ToString() объекта myObj — экземпляра класса MyComplexClass:

```
MyComplexClass myObj = new MyComplexClass();
Console.WriteLine(myObj.ToString());
```

Этот метод является одним из унаследованных от System.Object (на диаграмме для ясности не показан) и просто возвращает строковое имя класса объекта вместе с именами всех имеющих к нему отношение пространств имен.

Данное приложение ничего особенного не делает, но немного позже в данной главе оно будет использовано для демонстрации нескольких важных понятий и приемов.

Класс System.Object

Поскольку все классы являются потомками System.Object, они все могут обращаться как к защищенным, так и к общедоступным членам этого класса, потому ознакомиться с ними совершенно не помешает. Методы класса System.Object перечислены в табл. 9.2.

Эти методы являются базовыми методами, которые должны обязательно поддерживаться типами объектов в .NET Framework, хотя некоторые из них могут вам никогда не пригодиться (разве что в особых обстоятельствах, как, например, метод GetHashCode).

Метод GetType() может быть полезен при использовании полиморфизма, т.к. позволяет выполнять различные операции с объектами в зависимости от их типа, а не одну и ту же операцию для всех объектов. Например, при наличии функции, принимающей параметр типа object (это означает, что ей можно передавать практически все, что угодно), можно сделать так, чтобы для объектов определенного типа выполнялись дополнительные действия. Комбинация метода GetType и операции typeof (C#-операция, преобразующая имя класса в объект System.Type) позволяет выполнить такую операцию сравнения:

```
if (myObj.GetType() == typeof(MyComplexClass))
{
   // myObj является экземпляром класса MyComplexClass.
}
```

Возвращаемый объект System.Type способен и на многое другое. Также полезно переопределять метод ToString, особенно если содержимое объекта можно легко представить с помощью одной понятной для человека строки. Методы класса System.Object еще не раз будут встречаться в последующих главах, где и будет сообщаться дополнительная информация.

Конструкторы и деструкторы

При определении класса в C# часто не обязательно определять связанные с ним конструкторы и деструкторы, т.к. при их отсутствии компилятор добавляет их автоматически во время сборки кода. Однако при необходимости можно предоставлять собственные конструктор и/или деструктор и с их помощью выполнять, соответственно, инициализацию объектов и зачистку при их уничтожении.

Таблица 9.2. Методы класса `System.Object`

Метод	Возвращаемый тип	Виртуальный	Статический	Описание
`Object()`	–	Нет	Нет	Конструктор для типа `System.Object`. Автоматически вызывается конструкторами производных типов.
`~Object()` (известный также как `Finalize()` — о нем будет рассказано в следующем разделе)	–	Нет	Нет	Деструктор для типа `System.Object`. Автоматически вызывается деструкторами объектов производных типов; не может вызываться вручную.
`Equals(object)`	`bool`	Да	Нет	Сравнивает объект, для которого он был вызван, с другим объектом и возвращает `true`, если они равны. В реализации по умолчанию проверяет, указывает ли параметр объекта на тот же объект (ведь объекты являются ссылочными типами). Может переопределяться, если необходимо сравнивать объекты по-другому — например, чтобы сравнивались состояния двух объектов.
`Equals(object, object)`	`bool`	Нет	Да	Сравнивает два переданных ему объекта и проверяет их на эквивалентность. Эта проверка осуществляется с помощью метода `Equals(object)`. Если вместо обоих объектов указаны нулевые ссылки, возвращается `true`.
`ReferenceEquals (object, object)`	`bool`	Нет	Да	Сравнивает два переданных ему объекта и проверяет, указывают ли они на один и тот же экземпляр
`ToString()`	`string`	Да	Нет	Возвращает строку, соответствующую экземпляру объекта. По умолчанию это квалифицированное имя типа класса, но такое поведение можно переопределить и предоставить реализацию, более подходящую для данного типа класса.
`MemberwiseClone()`	`object`	Нет	Нет	Копирует объект, т.е. создает новый экземпляр того же типа и копирует в него члены исходного объекта. Копирование членов не приводит к созданию новых экземпляров этих членов. Все члены ссылочного типа в новом объекте указывают на те же объекты, что и в исходном классе. Этот метод является защищенным и поэтому может использоваться только в самом классе или его потомках.
`GetType()`	`System.Type`	Нет	Нет	Возвращает тип объекта в виде объекта `System.Type`.
`GetHashCode()`	`int`	Да	Нет	Используется в качестве хеш-функции для объектов, когда необходимо. Хеш-функция — это функция, которая возвращает значение, указывающее на состояние объекта в сжатой форме.

Простой конструктор можно добавить в класс с помощью такого синтаксиса:

```
class MyClass
{

   public MyClass()
   {
      // Код конструктора.
   }
}
```

Этот конструктор должен иметь то же имя, что и у класса, в котором он содержится, не принимать никаких параметров (что сделает его конструктором по умолчанию для данного класса) и быть общедоступным, чтобы с его помощью можно было создавать экземпляры объектов этого класса (см. главу 8).

Можно использовать и приватный конструктор по умолчанию, т.е. такой, который не позволяет создавать экземпляры данного класса (*не создаваемый* класс — см. главу 8):

```
class MyClass
{
   private MyClass()
   {
      // Код конструктора.
   }
}
```

И, наконец, аналогично в класс можно добавлять и конструкторы не по умолчанию, просто указав их параметры:

```
class MyClass
{
   public MyClass()
   {
      // Код конструктора по умолчанию.
   }
   public MyClass(int myInt)
   {
      // Код конструктора не по умолчанию (с параметром myInt).
   }
}
```

Количество конструкторов может быть любым (точнее — почти любым, ведь может не хватить памяти или возможных сочетаний параметров).

Деструкторы объявляются немного по-другому. Деструктор, используемый в .NET (и предоставляемый классом `System.Object`), называется `Finalize`, однако вместо переопределения `Finalize` используется следующий код:

```
class MyClass
{
   ~MyClass()
   {
      // Тело деструктора.
   }

}
```

То есть деструктор класса объявляется путем указания имени класса (как и для конструктора) с префиксом ~ (тильда). Код деструктора выполняется на этапе сборки мусора, позволяя освобождать ресурсы. После вызова деструктора выполняются неявные вызовы и деструкторов базовых классов, в том числе деструктора `Finalize` корневого класса `System.Object`. Это позволяет .NET Framework гарантировать их выполнение, но в слу-

чае переопределения деструктора `Finalize` вызовы деструкторов базовых классов придется осуществлять явно, что потенциально опасно (о вызовах методов базовых классов речь пойдет в следующей главе).

Последовательность выполнения конструкторов

Если в конструкторах класса необходимо решать несколько задач, то нужный для этого код удобно выделить в одно место и получить те же преимущества, что и при разбиении кода на функции (см. главу 6). Это можно сделать с помощью метода (как будет показано в главе 10), но в C# есть другой замечательный вариант: настроить конструктор так, чтобы перед выполнением своего собственного кода он вызывал какой-то другой конструктор.

Но вначале необходимо хорошо разобраться, что происходит по умолчанию при создании экземпляра класса. Этот вопрос интересен и сам по себе, а не только из-за централизации инициализирующего кода. Во время разработки объекты часто ведут себя не совсем так, как ожидается, из-за ошибок, которые возникают во время вызова конструкторов — обычно из-за неправильного создания экземпляра какого-то родительского класса в иерархии наследования или неверного предоставления информации конструктору базового класса. Понимание этого этапа жизненного цикла объекта может значительно упростить устранение подобных проблем.

Перед созданием экземпляра производного класса сначала должен быть создан экземпляр его базового класса, а для этого необходимо создать экземпляр базового класса того класса и т.д., вплоть до самого класса `System.Object.Object` (корневого для всех классов). Поэтому, какой бы конструктор не использовался для создания экземпляра класса, первым все равно вызывается конструктор `System.Object`.

Независимо от того, какой конструктор применяется в производном классе (по умолчанию или нет), для базового класса используется конструктор по умолчанию — если явно не указано иное поведение (каким образом — описано ниже). Проиллюстрируем последовательность выполнения на простом примере. Рассмотрим следующую иерархию объектов:

```
public class MyBaseClass
{
   public MyBaseClass()
   {
   }

   public MyBaseClass(int i)
   {
   }
}

public class MyDerivedClass : MyBaseClass
{
   public MyDerivedClass()
   {
   }

   public MyDerivedClass(int i)
   {
   }

   public MyDerivedClass(int i, int j)
   {
   }
}
```

В этом случае экземпляр класса `MyDerivedClass` создается так:

```
MyDerivedClass myObj = new MyDerivedClass();
```

Тогда происходит последовательность следующих событий:

- сначала выполняется конструктор `System.Object.Object`;
- затем выполняется конструктор `MyBaseClass.MyBaseClass`;
- и, наконец, выполняется конструктор `MyDerivedClass.MyDerivedClass`.

Но экземпляр класса `MyDerivedClass` может быть создан и так:

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

Тогда последовательность событий будет выглядеть по-другому:

- сначала выполняется конструктор `System.Object.Object`;
- затем выполняется конструктор `MyBaseClass.MyBaseClass`;
- и потом выполняется конструктор `MyDerivedClass.MyDerivedClass(int i)`.

А если создать экземпляр класса `MyDerivedClass` так:

```
MyDerivedClass myObj = new MyDerivedClass(4, 8);
```

то последовательность событий будет выглядеть следующим образом:

- сначала выполняется конструктор `System.Object.Object`;
- затем выполняется конструктор `MyBaseClass.MyBaseClass`;
- и, наконец, выполняется конструктор `MyDerivedClass.MyDerivedClass(int i, int j)`.

Такая система подходит для большинства случаев, но в некоторых ситуациях может потребоваться чуть больший контроль над происходящими событиями. Например, в случае применения последнего способа для создания экземпляра класса `MyDerivedClass` может быть необходимо, чтобы последовательность событий выглядела так:

- первым выполняется конструктор `System.Object.Object`;
- вторым выполняется конструктор `MyBaseClass.MyBaseClass(int i)`;
- третьим выполняется конструктор `MyDerivedClass.MyDerivedClass(int i, int j)`.

Тогда код, в котором используется параметр `int i`, можно поместить в `MyBaseClass(int i)`, и у конструктора `MyDerivedClass(int i, int j)` будет меньше работы: ему придется обрабатывать только параметр `int j`. (Здесь предполагается, что в обоих сценариях назначение параметра `int i` одинаково — на практике обычно так и бывает.) Язык C# позволяет при желании задавать подобное поведение.

Для этого предназначен так называемый *инициализатор конструктора* (constructor initializer) — код в определении метода после символа двоеточия. Например, в определении конструктора производного класса можно указать задействуемый конструктор базового класса:

```
public class MyDerivedClass : MyBaseClass
{
   ...
   public MyDerivedClass(int i, int j) : base(i)
   {
   }
}
```

Ключевое слово `base` указывает процессу создания экземпляров .NET использовать конструктор базового класса с заданными параметрами. В данном примере у него есть только один параметр — типа `int` (значение которого передается конструктору `MyDerivedClass` в параметре `i`), поэтому применяться будет метод `MyBaseClass(int i)`. Это означает, что `MyBaseClass` вызываться не будет, поэтому последовательность событий будет такой, какой нужно — а именно, такой, как описано перед данным примером.

Кстати, это ключевое слово позволяет указать литеральные значения для конструкторов базовых классов, например, в случае применения конструктора по умолчанию класса `MyDerivedClass` для вызова конструктора не по умолчанию класса `MyBaseClass`:

```
public class MyDerivedClass : MyBaseClass
{
  public MyDerivedClass() : base(5)
  {
  }
  ...
}
```

Это приведет к такой последовательности:

- первым выполняется конструктор `System.Object.Object`;
- вторым выполняется конструктор `MyBaseClass.MyBaseClass(int i)`;
- третьим выполняется конструктор `MyDerivedClass.MyDerivedClass()`.

Кроме ключевого слова `base`, имеется еще одно ключевое слово, которое можно применять в качестве инициализатора конструктора — `this`. Оно указывает процессу создания экземпляров .NET использовать перед вызовом указанного конструктора конструктор по умолчанию текущего класса:

```
public class MyDerivedClass : MyBaseClass
{
  public MyDerivedClass() : this(5, 6)
  {
  }
  ...
  public MyDerivedClass(int i, int j) : base(i)
  {
  }
}
```

В этом случае последовательность событий будет такой:

- выполняется конструктор `System.Object.Object`;
- выполняется конструктор `MyBaseClass.MyBaseClass(int i)`;
- выполняется конструктор `MyDerivedClass.MyDerivedClass(int i, int j)`;
- выполняется конструктор `MyDerivedClass.MyDerivedClass`.

Единственное ограничение состоит в том, что с помощью инициализатора конструктора можно указывать только один конструктор. Однако, как показано в предыдущем примере, это не такое уж серьезное ограничение, поскольку оно не мешает задавать довольно сложные последовательности выполнения.

Если инициализатор для конструктора не указан, то компилятор автоматически добавляет конструкцию `base()`. Это приводит к поведению по умолчанию, описанному в данном разделе.

Будьте осторожны: ошибка в определении конструкторов может привести к бесконечному циклу, например:

```
public class MyBaseClass
{
  publicMyBaseClass() : this(5)
  {
  }
```

```
    publicMyBaseClass(inti) : this()
    {
    }
}
```

Использование любого из этих конструкторов требует предварительного выполнения другого конструктора, который, в свою очередь, требует выполнения первого, и т.д. Этот код скомпилируется, но при попытке создать экземпляр `MyBaseClass` возникнет исключение `SystemOverflowException`.

Средства объектно-ориентированного программирования в VS и VCE

Поскольку ООП является фундаментальным аспектом .NET Framework, для облегчения процесса разработки ООП-приложений в VS и VCE предусмотрены специальные средства. Некоторые из них рассматриваются в настоящем разделе.

Окно Class View

В главе 2 было сказано, что окно Solution Explorer (Проводник решений) делит экранное пространство с еще одним окном — Class View (Представление классов). Оно отображает иерархию классов в приложении и позволяет сразу просматривать все характеристики используемых классов. На рис. 9.3 показано, как выглядит это окно для проекта из предыдущего практического занятия.

Окно имеет два главных раздела; нижний раздел содержит члены типов. Чтобы увидеть, как все происходит для нашего примера, а также действия, которые можно выполнять в окне Class View, необходимо выводить некоторые элементы, которые сейчас скрыты. Для этого отметьте в раскрывающемся списке Class View Grouping (Группировка представления классов) в верхней части окна Class View элементы, показанные на рис. 9.4.

Теперь отображаются члены классов и появится дополнительная информация (рис. 9.5).

Здесь могут использоваться разнообразные обозначения, в том числе и значки, перечисленные в табл. 9.3.

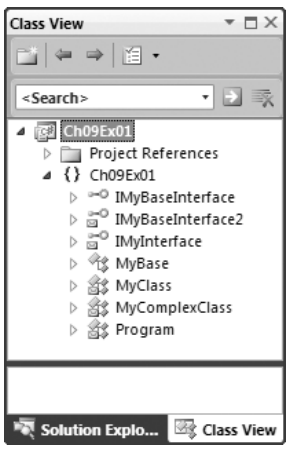


Рис. 9.3. *Внешний вид окна Class View*

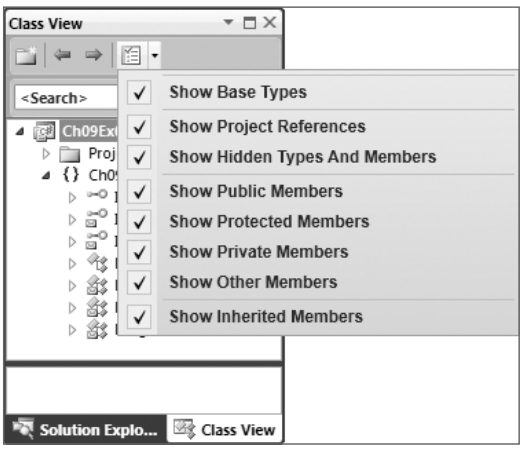


Рис. 9.4. *Настройка параметров Class View*

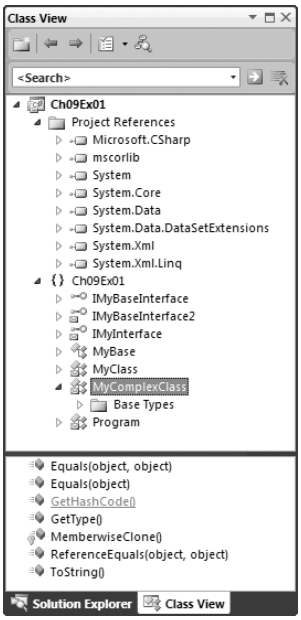


Рис. 9.5. *Отображение скрытых элементов и дополнительной информации в окне* Class View

Таблица 9.3. Некоторые обозначения из окна Class View

Значок	Что обозначает
	Проект
	Пространство имен
	Класс
	Интерфейс
	Метод
	Свойство
	Поле
	Структура
	Перечисление
	Элемент перечисления
	Событие
	Делегат
	Сборка

Некоторые из этих значков используются для определений не только классов, но и других типов, например, перечислений и структур. Под некоторыми элементами могут находиться и другие обозначения, указывающие на уровень доступа (под общедоступными элементами их нет). Эти обозначения приведены в табл. 9.4.

Таблица 9.4. Обозначения уровня доступа

Значок	Уровень доступа, на который он указывает
	Приватный
	Защищенный
	Внутренний

Символов для обозначения абстрактных, запечатанных и виртуальных элементов нет.

Помимо просмотра этой информации, здесь можно просмотреть код многих элементов. Двойной щелчок на элементе или щелчок правой кнопкой мыши и выбор в контекстном меню пункта Go To Definition (Перейти к определению) сразу переходит к тому месту в коде проекта, где содержится его определение (если оно доступно). Если код не доступен — как, например, для базовых типов (вроде `System.Object`) — тогда имеется вариант Browse Definition (Обзор определений) с переходом в окно Object Browser (Браузер объектов), которое будет описано в следующем разделе.

На рис. 9.5 имеется еще один узел — Project References (Ссылки проектов), который позволяет увидеть, на какие сборки ссылаются проекты. В данном случае к ним относятся еще и базовые типы .NET в сборках `mscorlib` и `system`, типы доступа к данным в `system.data` и типы для работы с XML в `system.xml`. Эти ссылки можно разворачивать и просматривать содержащиеся в сборках пространства имен и типы.

Для поиска вхождений типов и членов в коде нужно щелкнуть на элементе правой кнопкой мыши и выбрать в контекстном меню пункт Find All References (Найти все ссылки). Результаты поиска выводятся в окне Find Symbol Results (Результаты поиска символов) в нижней части экрана в виде вкладки в области Error List (Список ошибок). В окне Class View можно также переименовывать элементы. При этом есть возможность переименования ссылок на данный элемент во всех местах кода. Это позволяет исправлять опечатки в именах классов сколько угодно раз.

Кроме того, в VS 2010 появился новый способ перемещения по коду — он называется иерархией вызовов (Call Hierarchy) и доступен из окна Class View с помощью пункта View Call Hierarchy (Просмотреть иерархию вызовов) контекстного меню. Эта функция очень полезна для просмотра взаимодействия членов класса; она будет рассмотрена в следующей главе.

Окно Object Browser

Окно Object Browser (Браузер объектов) представляет собой расширенную версию окна Class View и позволяет просматривать другие доступные для проекта классы и даже полностью внешние классы. Оно может открываться как автоматически (пример приведен в предыдущем разделе), так и вручную, с помощью пункта меню View⇨Object Browser (Вид⇨Браузер объектов). Оно отображается в основном окне, где с ним можно работать точно так же, как и с окном Class View.

Это окно содержит ту же информацию, что и окно Class View, но с большим количеством типов .NET. Информация о выбранном элементе отображается в еще одном отдельном окне, как показано на рис. 9.6.

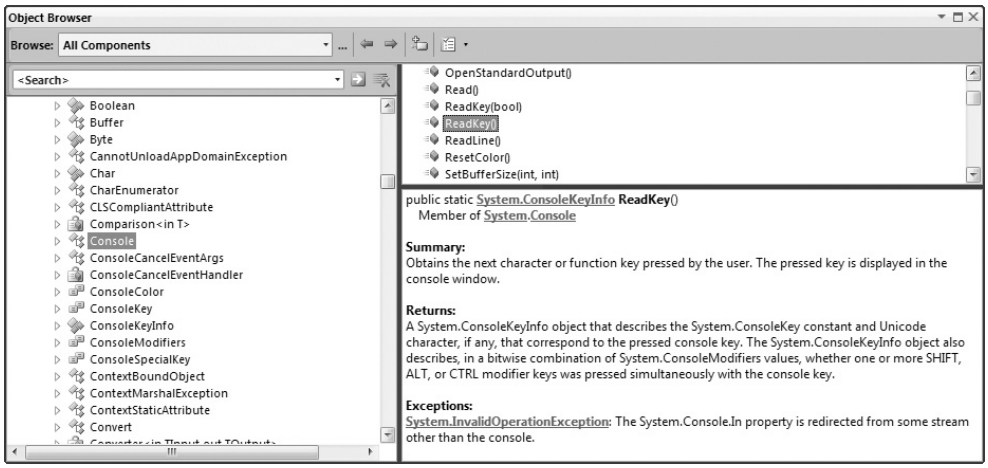


Рис. 9.6. *Выбор элемента в окне* Object Browser

Здесь выбран метод `ReadKey()` класса `Console` (этот класс находится в пространстве имен `System` в сборке `mscorlib`). В информационном окне справа внизу выводится сигнатура этого метода, класс, которому он принадлежит, и краткая информация о его функции. Все эти сведения могут быть полезны при изучении типов .NET, а также просто чтобы вспомнить, что делает тот или иной класс.

Помимо этого, данная информация полезна и для создаваемых типов. Внесите следующее изменение в код `Ch09Ex01`:

```
/// <summary>
/// В этом классе содержится моя программа!
/// </summary>
class Program
{
  static void Main(string[] args)
  {
    MyComplexClass myObj = new MyComplexClass();
    Console.WriteLine(myObj.ToString());
    Console.ReadKey();
  }
}
```

Фрагмент кода ***Ch09Ex01\Program.cs***

Вернитесь в окно Object Browser. Внесенное изменение проявится в информационном окне. Это пример XML-документации — данная тема выходит за рамки данной книги, но с ней не помешает ознакомиться самостоятельно.

Если вы вносили данное изменение вручную, то наверняка заметили, что после ввода трех слешей (///) IDE-среда самостоятельно добавляет большую часть остального кода. Она автоматически анализирует код, к которому применяется XML-документация, и создает элементарную XML-документацию — еще одно доказательство, что VS и VCE представляют собой просто замечательные инструменты для работы.

Добавление классов

В VS и VCE имеются средства, которые могут ускорить выполнение некоторых распространенных задач, часть которых относится и к ООП. Одним из них является мастер добавления новых элементов (Add New Item Wizard), который позволяет добавлять в проект новые классы с минимальным количеством ручной работы.

Для его вызова выберите пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент) или щелкните правой кнопкой мыши на проекте в окне Solution Explorer и выберите в контекстном меню нужный пункт. После этого появится диалоговое окно, в котором можно выбрать добавляемый элемент. Внешний вид этого окна в VS и VCE отличается, но функциональные возможности одинаковы. В любой IDE-среде для добавления класса необходимо выбрать в окне шаблонов элемент Class (Класс), как показано на рис. 9.7, указать имя файла, в котором будет содержаться данный класс, и щелкнуть на кнопке Add (Добавить). Созданному классу назначается имя, указанное для файла.

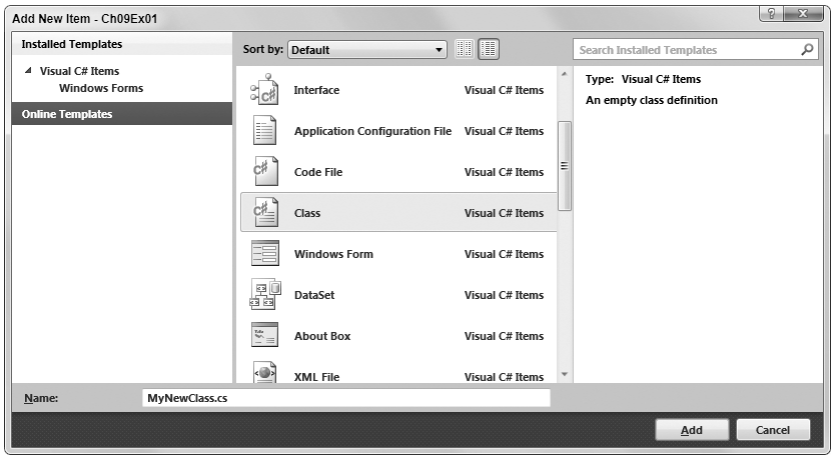


Рис. 9.7. *Добавление класса*

В предыдущем практическом занятии определения классов добавлялись вручную в файл `Program.cs`. Но зачастую размещение классов в отдельных файлах упрощает работу с ними. Если ввести информацию в окне Add New Item при открытом проекте `Ch09Ex01`, то в файле `MyNewClass.cs` будет сгенерирован следующий код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch09Ex01
{
   class MyNewClass
   {
   }
}
```

Этот класс — `MyNewClass` — определен в том же пространстве имен, что и класс точки входа `Program`, и поэтому его можно использовать в коде так, как если бы он был определен в том же самом файле. Здесь видно, что сгенерированный класс не содержит конструктор. Вспомните: если в определении класса нет конструктора, при компиляции кода компилятор самостоятельно добавляет конструктор по умолчанию.

Диаграммы классов

Одна из мощных функциональных возможностей среды VS, с которой вы еще не знакомы — генерация диаграмм классов и их применение для изменения проектов. Редактор диаграмм классов VS позволяет легко генерировать для кода UML-подобные диаграммы. Мы познакомимся с его работой в следующем практическом занятии, где будет сгенерирована диаграмма классов для ранее созданного проекта `Ch09Ex01`.

К сожалению, в VCE возможность создания диаграмм классов отсутствует, поэтому данное практическое занятие доступно только тем, у кого имеется VS.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Генерирование диаграммы классов

1. Откройте проект `Ch09Ex01`, созданный ранее в этой главе.
2. В окне Solution Explorer выделите файл `Program.cs` и щелкните на кнопке View Class Diagram (Просмотреть диаграмму классов) в панели инструментов, как показано на рис. 9.8.
3. После этого на экране появится диаграмма классов с именем `ClassDiagram1.cd`.
4. Щелкните на элементе `IMyInterface` и с помощью окна Properties (Свойства) измените значение его свойства `Position` на `Right`.

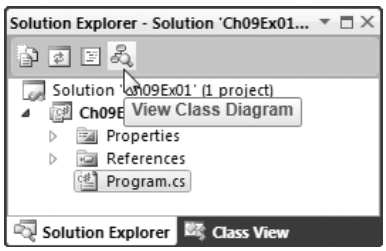


Рис. 9.8. *Кнопка View Class Diagram*

5. Щелкните правой кнопкой мыши на элементе `MyBase` и в появившемся контекстном меню выберите пункт Show Base Type (Показать базовый тип).
6. Передвиньте (перетаскиванием) объекты в диаграмме, чтобы она вам визуально нравилась. После этого диаграмма должна принять примерно такой вид, как показан на рис. 9.9.

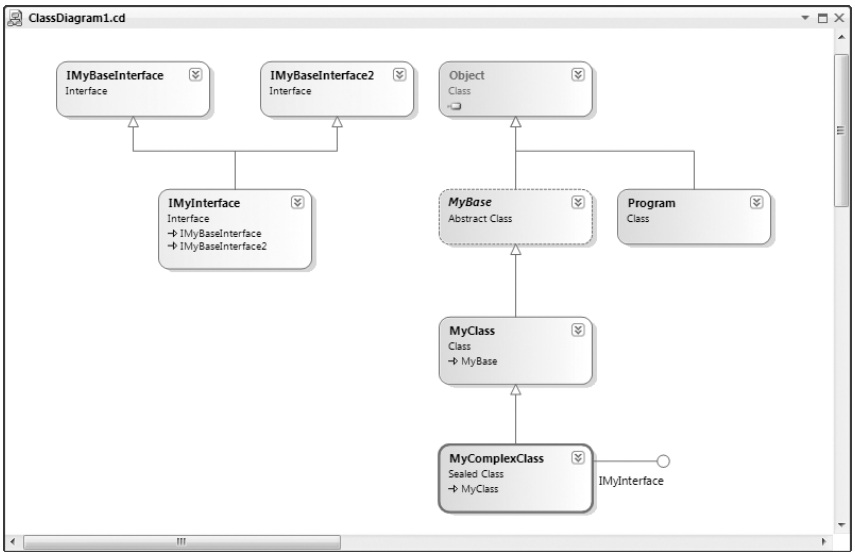


Рис. 9.9. *Конечный внешний вид диаграммы классов*

Описание работы

Приложив совсем небольшое количество усилий, вы создали диаграмму классов, аналогичную UML-диаграмме на рис. 9.2. Рассмотрим ее повнимательнее.

- Классы изображаются голубыми прямоугольниками с именем и типом.
- Интерфейсы изображаются зелеными прямоугольниками с именем и типом.
- Наследование изображается стрелками с белыми остриями (и иногда соответствующим текстом внутри прямоугольников классов).
- Классы, реализующие интерфейсы, имеют кружочек-выноску.
- Абстрактные классы обозначаются пунктирным контуром и курсивным именем.
- Запечатанные классы обозначаются жирным черным контуром.

Щелкнув на объекте, можно увидеть дополнительную информацию внутри окна Class Details (Информация о классе) в нижней части экрана (если этого окна нет, щелкните правой кнопкой на объекте и выберите пункт Class Details). Здесь можно просматривать (и изменять) члены класса. Изменять характеристики класса можно и в окне Properties (Свойства).

В главе 10 будет подробно рассказано, как добавлять в классы новые члены с помощью диаграммы классов.

С панели Toolbox (Элементы управления) в диаграмму можно добавлять новые элементы вроде классов, интерфейсов и перечислений и определять связи между ними. При этом код для добавляемых элементов генерируется автоматически.

Этот редактор позволяет графически проектировать целые семейства типов, вообще не пользуясь редактором кода. Конечно, добавлять конкретные функции придется вручную, но для начала совсем неплохо! В последующих главах мы еще вернемся к данному представлению и его возможностям. Но желающие могут изучить эти возможности и раньше.

Проекты библиотек классов

Классы можно размещать не только в отдельных файлах внутри проекта, но и в совершенно отдельных проектах. Проект, содержащий только классы (вместе с определениями нужных типов, но без входной точки), называется *библиотекой классов*.

Проекты библиотек классов компилируются в `.dll`-сборки, а обращаться к их содержимому можно с помощью ссылок на них из других проектов (которые могут быть, а могут и не быть частью того же решения). Это расширяет границы обеспечиваемой объектами инкапсуляции: библиотеки классов можно пересматривать и обновлять, не касаясь тех проектов, в которых они используются. Все это позволяет легко изменять предоставляемые классами функции (которые могут влиять на работу множества пользующихся ими приложений).

В следующем примере будет создан проект библиотеки классов и другой проект, в котором применяются содержащиеся в нем классы.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование библиотеки классов

1. Создайте новый проект типа Class Library (Библиотека классов) с именем `Ch09ClassLib` и сохраните его в каталоге `C:\BegVCSharp\Chapter09`, как показано на рис. 9.10.

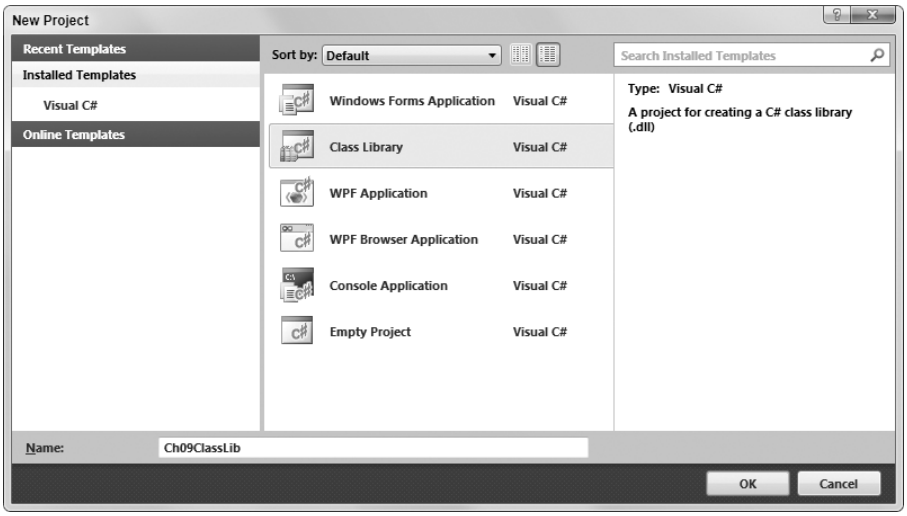


Рис. 9.10. *Создание нового проекта типа* Class Library

2. Измените имя файла `Class1.cs` на `MyExternalClass.cs`: в окне Solution Explorer щелкните на этом файле правой кнопкой мыши и выберите в контекстном меню пункт Rename (Переименовать). В открывшемся диалоговом окне щелкните на кнопке Yes (Да).
3. Код в файле `MyExternalClass.cs` автоматически изменится, чтобы соответствовать новому имени класса:

```
class MyExternalClass
{
}
```

Фрагмент кода ***Ch09ClassLib\MyExternalClass.cs***

4. Добавьте в проект новый класс с именем файла `MyInternalClass.cs`.
5. Явно объявите класс `MyInternalClass` внутренним:

```
internal class MyInternalClass
{
}
```

Фрагмент кода ***Ch09ClassLib\MyExternalClass.cs***

6. Скомпилируйте проект. У него нет точки входа, поэтому запустить его как обычный проект невозможно — просто выберите пункт меню Build⇨Build Solution (Сборка⇨Собрать решение).
7. Создайте новый проект типа Console Application (Консольное приложение), назовите его `Ch09Ex02` и сохраните в каталоге `C:\BegVCSharp\Chapter09`.
8. Выберите пункт меню Project⇨Add Reference (Проект⇨Добавить ссылку) или щелкните правой кнопкой мыши в окне Solution Explorer на папке References (Ссылки) и выберите в контекстном меню пункт с таким же названием.
9. Перейдите на вкладку Browse (Обзор), найдите каталог `C:\BegVCSharp\Chapter09\Chapter09\Ch09ClassLib\bin\Debug\` и дважды щелкните на файле `Ch09ClassLib.dll`.

10. По завершении операции удостоверьтесь, что соответствующая ссылка была добавлена в окно Solution Explorer (рис. 9.11).
11. Откройте окно Object Browser и разверните добавленную ссылку, чтобы увидеть, какие объекты содержатся в ней (рис. 9.12).
12. Измените код в файле `Program.cs` следующим образом:

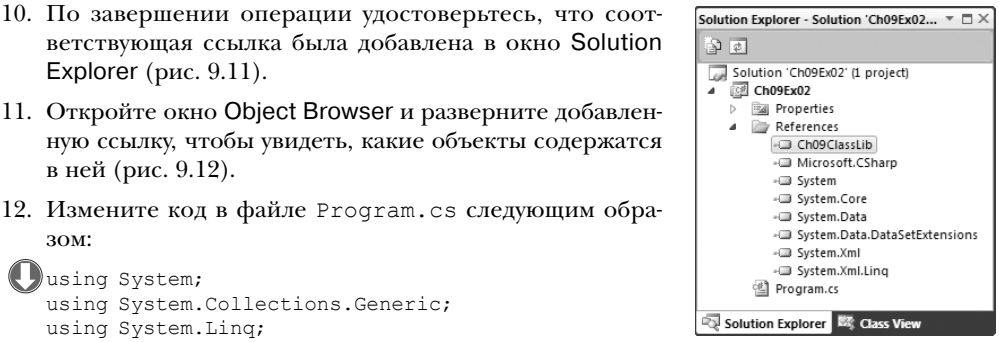


Рис. 9.11. *Ссылки в окне Solution Explorer*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch09ClassLib;

namespace Ch09Ex02
{
   class Program
   {
      static void Main(string[] args)
      {
         MyExternalClass myObj = new MyExternalClass();
         Console.WriteLine(myObj.ToString());
         Console.ReadKey();
      }
   }
}
```

Фрагмент кода ***Ch09Ex02\Program.cs***

13. Запустите приложение. На рис. 9.13 показан результат, который должен получиться.

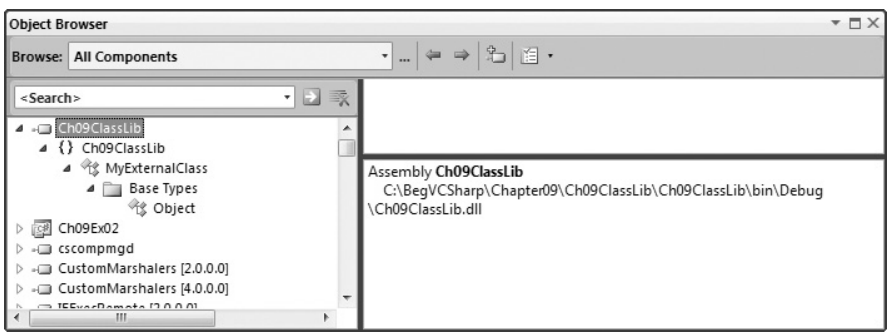


Рис. 9.12. *Содержимое новой ссылки в окне Object Browser*

Рис. 9.13. *Приложение `Ch09Ex02` в действии*

Описание полученных результатов

В этом примере были созданы два проекта: проект библиотеки классов и проект консольного приложения. Первый называется `Ch09ClassLib` и содержит два класса: общедоступный `MyExternalClass` и `MyInternalClass`, который доступен только внутри проекта. Класс `MyInternalClass` после создания по умолчанию был неявным, поскольку не имел никакого модификатора доступа. Однако уровень доступности рекомендуется указывать явно, поскольку это делает код более удобочитаемым, поэтому и было добавлено ключевое слово `internal`. Проект консольного приложения `Ch09Ex02` содержит простой код, использующий проект библиотеки классов.

Приложение, в котором используются классы, определенные во внешней библиотеке, часто называют клиентским приложением библиотеки, а код, в котором используется определенный разработчиком класс — соответственно, клиентским кодом.

Чтобы использовать классы из `Ch09ClassLib`, в консольное приложение была добавлена ссылка на `Ch09ClassLib.dll`. В данном примере для этого достаточно было указать на выходной файл библиотеки классов, хотя не менее легко можно было бы скопировать этот файл в какую-то локальную папку `Ch09Ex02` — после этого можно было бы продолжать разработку библиотеки классов без влияния на само консольное приложение. Тогда для замены старой версии сборки новой было бы достаточно просто скопировать сгенерированный DLL-файл вместо старого.

После добавления ссылки вы просмотрели доступные файлы в окне Object Browser. Поскольку класс `MyInternalClass` является внутренним, его в этом окне не видно: он недоступен для внешних проектов. Но класс `MyExternalClass` доступен для внешних проектов, и именно он и применяется в консольном приложении.

Можно попробовать заменить код в консольном приложении кодом, использующим внутренний класс:

```
static void Main(string[] args)
{
  MyInternalClass myObj = new MyInternalClass();
  Console.WriteLine(myObj.ToString());
  Console.ReadKey();
}
```

Но при попытке скомпилировать этот код компилятор выдаст сообщение об ошибке:

```
'Ch09ClassLib.MyInternalClass' is inaccessible due to its protection level
Ch09ClassLib.MyInternalClass недоступен из-за его уровня защиты
```

Такой прием использования классов из внешних сборок является основой программирования на C# и в .NET Framework. Вообще-то именно так и используются любые классы из .NET Framework.

Интерфейсы или абстрактные классы

В этой главе уже было показано, как создавать интерфейсы и абстрактные классы (хотя пока и без членов, об этом пойдет речь в главе 10). Эти два типа во многом похожи, поэтому стоит разобраться, в каких случаях лучше использовать интерфейс, а в каких — абстрактный класс.

Сначала поговорим о сходствах. И абстрактные классы, и интерфейсы могут иметь члены, наследуемые производными классами. Ни интерфейсы, ни абстрактные классы не позволяют создавать непосредственные экземпляры, но позволяют объявлять переменные

этих типов. При этом таким переменным можно с помощью полиморфизма присваивать порожденные от этих типов объекты. В любом случае можно использовать члены этих типов через такие переменные, хотя и без непосредственного доступа к другим членам производного объекта.

А теперь об отличиях. Производные классы могут наследоваться только от единственного базового класса, а это значит, что прямое наследование возможно только от одного абстрактного класса (хотя в цепочке наследования может быть несколько абстрактных классов). Однако классы могут использовать любое количество интерфейсов, хотя это несущественное различие: схожих результатов можно достичь и в том, и в другом случае. Просто подход с интерфейсами немного другой.

Абстрактные классы могут иметь как *абстрактные члены* (без кода, они должны быть реализованы в производном классе, если только тот сам не является абстрактным), так и *не абстрактные члены* (реализованные в коде класса, могут быть виртуальными с возможностью переопределения в производном классе). А *члены интерфейсов* должны быть реализованы в классе, использующем данный интерфейс, но своего кода не имеют. Далее, члены интерфейсов по определению являются общедоступными (поскольку предназначены для внешнего применения), а члены абстрактных классов могут быть и приватными (если они не абстрактные), защищенными, внутренними или защищенными внутренними (тогда они доступны только из кода внутри приложения либо из производного класса). Кроме того, интерфейсы не могут содержать ни полей, ни конструкторов, ни деструкторов, ни статических членов, ни констант.

Абстрактные классы предназначены служить в качестве базовых классов для объектов с общими ключевыми характеристиками — например, с общим назначением и структурой. Интерфейсы же предназначены для использования классами, которые отличаются на более фундаментальном уровне, но при этом все-таки имеют что-то общее.

Например, рассмотрим семейство объектов, представляющих поезда. В базовом классе — `Train` (Поезд) — должно содержаться основное определение поезда, вроде информации о ширине колеи и типе локомотива (паровой, дизельный и т.д.). Однако такой класс должен быть абстрактным, поскольку не бывает поезда "вообще". Для создания "реального" поезда нужно добавить характеристики, специфические для конкретного поезда — например, создать производные классы вроде `PassengerTrain` (Пассажирский поезд), `FreightTrain` (Товарный поезд) и `424DoubleBogey` (Двухосный поезд 424), как показано на рис. 9.14.

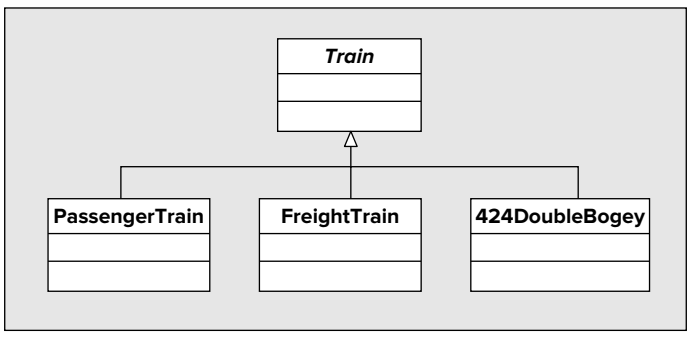


Рис. 9.14. *Диаграмма классов, представляющих поезда*

Аналогично можно определить и семейство объектов-автомобилей, с абстрактным базовым классом `Car` (Автомобиль) и производными классами вроде `Compact` (Малолитражка),

SUV (Внедорожник) и PickUp (Пикап). Классы Car и Train могут даже иметь общий базовый класс, например, Vehicle (Транспорт), как показано на рис. 9.15.

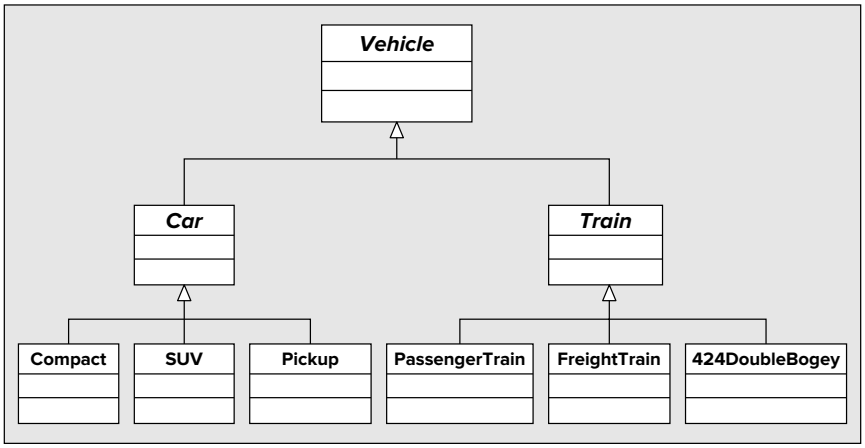


Рис. 9.15. *Диаграмма классов, представляющих транспортные средства*

Некоторые из классов ниже в иерархии могут иметь общие характеристики по своему предназначению, а не по родительскому классу. Например, классы PassengerTrain, Compact, SUV и Pickup могут перевозить пассажиров и поэтому могут обладать общим интерфейсом IPassengerCarrier (Пассажирский транспорт). Классы FreightTrain и PickUp могут перевозить тяжелые грузы и тоже могут тоже иметь общий интерфейс — IHeavyLoadCarrier (Грузовой транспорт), как показано на рис. 9.16.

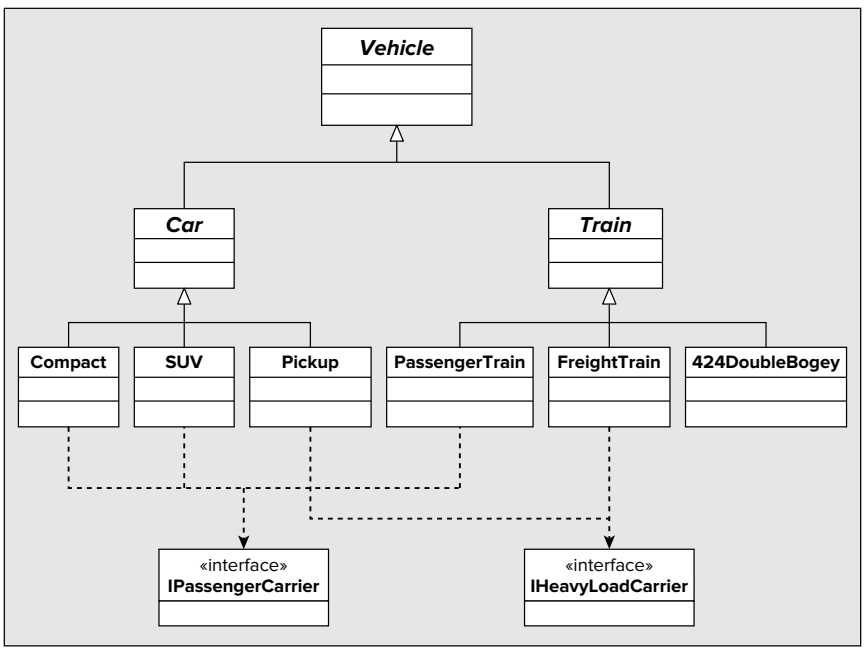


Рис. 9.16. *Интерфейсы* `IPassengerCarrier` *и* `IHeavyLoadCarrier`

Разбивая подобным образом системы объектов перед назначением конкретных деталей, можно четко видеть, где лучше использовать абстрактные классы, а где интерфейсы. Результат в этом примере невозможно было бы получить с помощью только интерфейсов или только наследования от абстрактных классов.

Типы-структуры

В главе 8 было сказано, что структуры и классы очень похожи, но структуры представляют собой типы значения, а классы — ссылочные типы. Что это на самом деле означает? Легче объяснить все это на конкретном примере.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Классы или структуры

1. Создайте новый проект консольного приложения с именем `Ch09Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter09`.
2. Измените его код следующим образом:

```
namespace Ch09Ex03
{
  class MyClass
  {
    public int val;
  }

  struct myStruct
  {
    public int val;
  }

  class Program
  {
    static void Main(string[] args)
    {
      MyClass objectA = new MyClass();
      MyClass objectB = objectA;
      objectA.val = 10;
      objectB.val = 20;
      myStruct structA = new myStruct();
      myStruct structB = structA;
      structA.val = 30;
      structB.val = 40;
      Console.WriteLine("objectA.val = {0}", objectA.val);
      Console.WriteLine("objectB.val = {0}", objectB.val);
      Console.WriteLine("structA.val = {0}", structA.val);
      Console.WriteLine("structB.val = {0}", structB.val);
      Console.ReadKey();
    }
  }
}
```

Фрагмент кода ***Ch09Ex03\Program.cs***

3. Запустите приложение. На рис. 9.17 показан результат, который должен получиться.

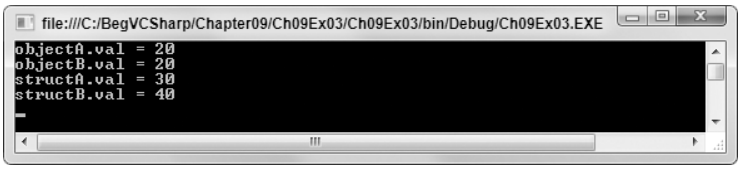


***Рис. 19.17**. Приложение `Ch09Ex03` в действии*

Описание работы

Рассматриваемое в примере приложение содержит два определения типов: определение структуры `myStruct`, имеющей одно общедоступное поле `val`, и определение класса `MyClass` с таким же полем (о полях-членах классов будет рассказано в главе 10; пока достаточно знать, что синтаксис выглядит одинаково). Далее для экземпляров обоих этих типов выполняются следующие одинаковые операции.

1. Объявляется переменная данного типа.
2. Для этой переменной создается новый экземпляр данного типа.
3. Объявляется вторая переменная такого же типа.
4. Второй переменной присваивается первая переменная.
5. Присваивается значение полю `val` в первой переменной.
6. Присваивается значение полю `val` во второй переменной.
7. Выводятся значения полей `val` обеих переменных.

Для переменных обоих типов выполняются одинаковые операции, но результаты их выполнения различны. При выводе значения поля `val` типы-объекты имеют одинаковые значения, а типы-структуры — разные. Почему?

Дело в том, что объекты представляют собой *ссылочные* типы. Когда переменной присваивается объект, фактически этой переменной присваивается просто *указатель* на объект, на который она затем ссылается. Указатель в коде на самом деле является адресом в памяти. В данном случае адрес указывает на место в памяти, где находится объект. При присваивании первой ссылки на объект второй переменной типа `MyClass` с помощью показанной ниже строки кода фактически выполняется копирование этого адреса:

```
MyClass objectB = objectA;
```

И после этого в обеих переменных содержатся указатели на один и тот же объект.

Но структуры представляют собой *типы значения*. Переменная содержит не указатель на структуру, а саму структуру. Поэтому при присваивании первой структуры второй переменной типа `myStruct` с помощью показанной ниже строки фактически выполняется копирование всей информации из одной структуры в другую:

```
myStruct structB = structA;
```

Подобное поведение уже обсуждалось ранее в книге при рассмотрении простых типов переменных вроде `int`. В результате две переменные типа структуры содержат различные структуры. Процесс применения указателей скрыт от разработчика в управляемом коде C#, что делает его проще.

Небезопасный код позволяет выполнять в C# низкоуровневые операции для обработки указателей, но эта более сложная тема не будет здесь рассматриваться.

Поверхностное копирование и глубокое копирование

Копирование объектов из одной переменной в другую по значению, а не по ссылке (т.е. подобно структурам) может оказаться довольно сложной задачей. Поскольку один объект может содержать ссылки на множество других объектов, вроде полей и т.д., такое копирование может потребовать сложной обработки. Простого копирования каждого члена из одного объекта в другой может быть недостаточно, поскольку некоторые из этих членов могут сами ссылаться на какие-то другие типы.

В .NET Framework все это учтено. Простое почленное копирование объектов можно выполнить с помощью метода `MemberwiseClone`, унаследованного от класса `System.Object`. Этот метод является защищенным, но можно объявить общедоступный метод для вызывающего его объекта. Такой способ называется *поверхностным копированием* (shallow copy), т.к. не учитывает члены ссылочных типов. Это означает, что члены ссылочных типов в новом объекте ссылаются на те же объекты, что и эквивалентные им члены в исходном объекте, и во многих случаях такое поведение неприемлемо. Если нужно, чтобы новые экземпляры создавались с копированием значений (а не только ссылок), необходимо *глубокое копирование* (deep copy).

Существует один интерфейс, который можно реализовать и который позволяет делать это стандартным образом, называется он `ICloneable`. При применении этого интерфейса нужно реализовать единственный метод, который он содержит — `Clone`. Этот метод возвращает значение типа `System.Object`. Для получения такого объекта, т.е. реализации тела метода, допустимы любые действия. Это означает, что при необходимости можно реализовать глубокое копирование (но если необходимости нет, можно ограничиться и поверхностным копированием). Более подробно об этом будет рассказано в главе 11.

Резюме

В этой главе было показано, как определять классы и интерфейсы в C# и претворить в жизнь теорию из предыдущей главы. Вы познакомились с синтаксисом C# для создания базовых определений, с имеющимися ключевыми словами для указания уровня доступности, со способами наследования от интерфейсов и других классов, с тем, как определять абстрактные и запечатанные классы для управления таким наследованием, а также со способами определения конструкторов и деструкторов.

Далее был рассмотрен класс `System.Object` — корневой базовый класс для любого определяемого класса. Этот класс предоставляет несколько методов, часть которых являются виртуальными, что позволяет переопределять их реализации. Данный класс позволяет считать экземпляр любого объекта экземпляром данного типа, а это позволяет применять полиморфизм с любым объектом.

Затем были описаны некоторые средства, имеющиеся в VS и VCE для разработки приложений на базе ООП — окна Class View и Object Browser и быстрый способ добавления в проект новых классов. Кроме того, было показано, как создавать сборки, не предназначенные для выполнения, но содержащие определения классов для использования в других проектах.

После этого было более подробно рассказано об абстрактных классах и интерфейсах, об их сходствах и отличиях и о том, когда лучше применять те и другие.

Напоследок снова была затронута тема ссылочных типов и типов значения, но на этот раз в связи со структурами (типы значения, эквивалентные объектам). Это привело к рассмотрению поверхностного и глубокого копирования объектов, о которых еще будет рассказываться в настоящей книге.

Следующая глава посвящена определению членов классов (таких как свойства и методы), которые поднимут уровень вашего владения ООП в C# до необходимого для разработки реальных приложений.

Упражнения

1. Что неверно в следующем коде?

```
public sealed class MyClass
{
   // Члены класса.
}

public class myDerivedClass : MyClass
{
   // Члены класса.
}
```

2. Как определяются не создаваемые классы?
3. Зачем нужны не создаваемые классы? Как пользоваться их возможностями?
4. Создайте проект библиотеки классов с именем `Vehicles` и напишите в нем код, реализующий семейство описанных в этой главе классов `Vehicle`. Нужно реализовать девять классов и два интерфейса.
5. Создайте проект консольного приложения с именем `Traffic`, ссылающийся на проект `Vehicles.dll` (из упражнения 4). Добавьте в него функцию `AddPasenger`, принимающую любой объект с интерфейсом `IPassengerCarrier`. Для проверки работоспособности кода вызовите эту функцию для экземпляров всех объектов, которые поддерживают этот интерфейс, и выводите результаты с помощью метода `ToString`, унаследованного от класса `System.Object`.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Определения классов и интерфейсов	Классы определяются с помощью ключевого слова `class`, а интерфейсы — с помощью ключевого слова `interface`. Ключевые слова `public` и `internal` позволяют задать уровень доступности класса и интерфейса, а для управления наследованием классов имеются дополнительные ключевые слова `abstract` и `sealed`. Родительские классы и интерфейсы указываются через запятую в списке после двоеточия за именем класса или интерфейса. В определении класса разрешается указать только один родительский класс, и тогда он должен быть первым элементом списка.
Конструкторы и деструкторы	Классы снабжаются готовыми реализациями конструкторов по умолчанию и деструкторов. Собственные деструкторы приходится создавать лишь в редких случаях. При определении конструктора можно указать его доступность, имя класса и возможные параметры. Конструкторы базовых классов выполняются перед конструкторами производных классов; внутри класса можно управлять последовательностью выполнения с помощью ключевых слов `this` и `base`.
Библиотеки классов	Можно создавать проекты библиотек классов, которые содержат только определения классов. Эти проекты нельзя выполнять непосредственно, доступ к ним возможен только из клиентского кода в выполняемом приложении. В VS и VCE имеются различные средства для создания, изменения и просмотра классов.

Тема	Основные концепции
Семейства классов	Классы можно группировать в семейства, которые обладают сходным поведением или общими характеристиками. Это достигается наследованием от общего базового класса (который может быть абстрактным) или реализацией какого-то интерфейса.
Определения структур	Структуры определяются аналогично классам, но не забывайте, что структуры являются типами-значениями, а классы — ссылочными типами.
Копирование объектов	При копировании объектов необходимо учитывать, что внутри могут содержаться другие объекты, для которых не всегда годится копирование указывающих на них ссылок. Копирование ссылок называется поверхностным копированием, а полное копирование информации — глубоким копированием. Для обеспечения возможности глубокого копирования в определении класса можно использовать интерфейс `ICloneable`.

10

Определение членов классов

В ЭТОЙ ГЛАВЕ...

- Определение членов классов
- Применение диаграммы классов для добавления членов
- Управление наследованием членов классов
- Определение вложенных классов
- Реализация интерфейсов
- Использование частичных определений классов

В этой главе рассматривается определение членов в классах — полей, свойств и методов. Сначала будет показано, какой код должен использоваться для определения членов каждого из этих типов, а затем — как генерировать структуру этого кода с помощью мастеров, а также быстро изменять код членов за счет редактирования их свойств.

После рассмотрения основных приемов по определению членов в классах вы узнаете о некоторых более совершенных приемах, таких как сокрытие членов базового класса, вызов переопределенных членов базового класса, вложенные определения типов и частичные определения классов.

И, наконец, в главе предоставляется возможность применить изученную теорию на практике за счет создания простой библиотеки классов, которая будет дорабатываться и использоваться в последующих главах.

Определение членов

Внутри определения класса предоставляются определения всех его членов, а именно — полей, методов и свойств. Каждый из этих членов обладает собственным уровнем доступности, который во всех случаях указывается с помощью одного из следующих ключевых слов.

- `public` (общедоступный). Члены доступны из любого кода.
- `private` (приватный). Члены доступны только из кода, который является частью данного класса (используется по умолчанию, если ключевое слово вообще не указано).
- `internal` (внутренний). Члены доступны только внутри сборки (проекта), в которой они определены.
- `protected` (защищенный). Члены доступны только из кода, который является частью либо данного, либо производного класса.

Два последних ключевых слова могут указываться вместе (`protected internal`), что позволяет делать члены доступными только из классов, которые являются производными в рамках кода данного проекта (точнее — внутри сборки).

Поля, методы и свойства могут объявляться с использованием ключевого слова `static`, которое превращает их в статические члены, принадлежащие классу, а не экземплярам, как объяснялось в главе 8.

Определение полей

Поля определяются с применением стандартного синтаксиса объявления переменных (и, при желании, инициализации) вместе с перечисленными выше модификаторами:

```
class MyClass
{
  public int MyInt;
}
```

При назначении имен общедоступным (`public`) полям в .NET Framework используется формат PascalCasing, а не camelCasing, и здесь применяется именно такой формат именования. Поэтому имя поля в приведенном выше примере выглядит как `MyInt`, а не `myInt`. Такая схема именования не является обязательной, но ее имеет смысл придерживаться. Для приватных (`private`) полей никаких особых рекомендаций не существует и обычно применяется формат camelCasing.

При определении полей также может использоваться ключевое слово `readonly`, которое означает, что данному полю значение может присваиваться только либо во время выполнения кода конструктора, либо во время операции начального присваивания:

```
class MyClass
{
  public readonly int MyInt = 17;
}
```

Как уже упоминалось во вводном разделе настоящей главы, поля могут объявляться статическими с применением ключевого слова `static`:

```
class MyClass
{
  public static int MyInt;
}
```

К статическим полям доступ может получаться только через класс, в котором они определены (в предыдущем примере — `MyClass.MyInt`), а не через экземпляры объектов этого класса. Ключевое слово `const` позволяет создавать константные значения. Члены `const` являются статическими по определению, поэтому в их случае указывать модификатор `static` не нужно (на самом деле это считается ошибкой).

Определение методов

Методы определяются с применением стандартного синтаксиса определения функций вместе с модификаторами доступности и необязательным модификатором `static`, как показано в следующем примере:

```
class MyClass
{
  public string GetString()
  {
    return "Here is a string.";
  }
}
```

Как и для общедоступных полей, для имен общедоступных методов в .NET Framework используется формат PascalCasing.

Следует запомнить, что указание ключевого слова `static` делает возможным доступ к этому методу только через класс, в котором он определен, но не через экземпляры объектов этого класса. При определении методов можно также использовать следующие ключевые слова.

- `virtual` (виртуальный). Метод может быть переопределен.
- `abstract` (абстрактный). Метод должен быть обязательно переопределен в не абстрактных производных классах (это ключевое слово допустимо только в абстрактных классах).
- `override` (переопределенный). Метод переопределяет метод из базового класса (это ключевое слово должно обязательно использоваться при переопределении метода).
- `extern` (внешний). Определение данного метода находится в каком-то другом месте.

Ниже приведен пример переопределения метода:

```
public class MyBaseClass
{
```

```
  public virtual void DoSomething()
  {
    // Базовая реализация.
  }
}
public class MyDerivedClass : MyBaseClass
{
  public override void DoSomething()
  {
    // Реализация в производном классе, переопределяющая базовую реализацию.
  }
}
```

Вместе с `override` можно также использовать ключевое слово `sealed` (запечатанный), которое указывает, что данный метод не может быть модифицирован в производных классах, т.е. переопределяться он больше не будет. Ниже показан пример:

```
public class MyDerivedClass : MyBaseClass
{
  public override sealed void DoSomething()
  {
     // Реализация в производном классе, переопределяющая базовую реализацию.
  }
}
```

С использованием ключевого слова `extern` можно предоставить реализацию метода за пределами проекта, но эта тема является сложной и здесь не рассматривается.

Определение свойств

Свойства определяются сходным с полями образом, но имеют больше особенностей. Как уже упоминалось ранее, свойства являются более сложными, чем поля, поскольку могут выполнять дополнительную обработку перед изменением состояния — и, на самом деле, могут вообще не изменять состояния. Это удается благодаря наличию двух похожих на функции блоков, один из которых отвечает за получение значения свойства, а второй — за его установку.

Эти блоки, также называемые *средствами доступа* (accessors), определяются с помощью ключевых слов `get` и `set` и могут использоваться для управления уровнем доступа к свойству. Указание только одного из этих ключевых слов позволяет создавать свойства, доступные только для записи или только для чтения (отсутствие блока `get` дает свойство, доступное только для записи, а отсутствие блока `set` — свойство, доступное только для чтения). Разумеется, это касается только внешнего кода, поскольку код внутри класса имеет доступ к тем же данным, что и эти блоки кода. Вместе с этими блоками допускается использовать модификаторы доступности, делая, например, блок `get` общедоступным (`public`), а блок `set` — защищенным (`protected`). Для получения допустимого свойства в код должен быть обязательно включен хотя бы один из этих блоков (очевидно, что от свойства, которое нельзя ни считывать, не изменять, будет мало толку).

Базовая структура свойства предусматривает указание сначала стандартного ключевого слова, отражающего уровень доступности (наподобие `public`, `private` и т.д.), следом за ним имени типа, затем имени свойства и, наконец, одного или обоих блоков `get` и `set` с кодом для обработки свойства внутри:

```
public int MyIntProp
{
  get
  {
    // Код для получения значения свойства.
  }
```

```
  set
  {
    // Код для установки значения свойства.
  }
}
```

Для имен общедоступных свойств в .NET тоже используется формат Pascal-Casing, а не camelCasing.

Первая строка в определении свойства представляет собой фрагмент, который очень похож на определение поля. Отличие состоит лишь в том, что никакой точки с запятой в конце этой строки нет, вместо этого там идет блок кода, содержащий вложенные блоки `get` и `set`.

В блоках `get` должно обязательно присутствовать возвращаемое значение, имеющее тип свойства. Простые свойства часто ассоциируются с одним приватным полем, управляющим доступом к нему, в случае которого в блоке `get` может возвращаться значение непосредственно самого поля:

```
// Поле, используемое свойством.
private int myInt;

// Свойство.
public int MyIntProp
{
  get
  {
    return myInt;
  }
  set
  {
    // Код, отвечающий за установку значения для свойства.
  }
}
```

Код, находящийся за пределами класса, не сможет получать доступ к данному полю `myInt` напрямую из-за того, что уровень доступности последнего был указан как `private`. Вместо этого внешнему коду для доступа к этому полю придется использовать свойство. Функция `set` позволяет присваивать значение полю похожим образом. В ней для ссылки на значение, получаемое от пользователя свойства, применяется ключевое слово `value`:

```
// Поле, используемое свойством.
private int myInt;

// Свойство.
public int MyIntProp
{
  get
  {
    return myInt;
  }
  set
  {
    myInt = value;
  }
}
```

`value` эквивалентно значению того же типа, что и у свойства, поэтому если свойство имеет такой же тип, как у поля, необходимость заботиться о приведении типов не будет возникать никогда.

Данное простое свойство всего лишь защищает от прямого доступа к полю myInt. Однако реальная мощь свойств проявляется, когда необходим немного больший контроль. Например, можно было бы реализовать блок set следующим образом:

```
set
{
  if (value >= 0 && value <= 10)
      myInt = value;
}
```

Здесь myInt изменяется только в случае присваивания свойству значения в диапазоне между 0 и 10. В ситуациях, подобных этой, должен будет сделан важный выбор относительно того, что должно происходить в случае использования недействительного значения. Существует четыре возможных варианта:

- ничего (как и в предыдущем коде);
- полю должно присваиваться значение по умолчанию;
- выполнение кода должно продолжаться так, будто ничего страшного не произошло, но с регистрацией события в журнале для последующего анализа причин его возникновения;
- должно генерироваться исключение.

В целом наиболее предпочтительными считаются два последних варианта. Какой из них будет лучше, зависит от типа класса и уровня контроля, необходимого пользователям класса. Вариант с генерацией исключения предоставляет пользователям класса довольно высокий контроль, позволяя им знать о том, что происходит, и надлежащим образом реагировать. Для обеспечения такого поведения можно воспользоваться одним из стандартных исключений, которые предлагаются в пространстве имен System:

```
set
{
  if (value >= 0 && value <= 10)
    myInt = value;
  else
    throw (new ArgumentOutOfRangeException("MyIntProp", value,
           "MyIntProp must be assigned a value between 0 and 10."));
      // MyIntProp может присваиваться только значение в диапазоне от 0 до 10
}
```

Исключение обрабатывается за счет помещения в код, где используется соответствующее свойство, логики try...catch...finally, как было показано в главе 7.

Регистрация данных, например, в текстовом файле, тоже может оказаться удобным вариантом, скажем, в производственном коде, где проблемы на самом деле не должны возникать. Это позволяет разработчикам проверять показатели производительности и при необходимости даже отлаживать существующий код.

Со свойствами, как и с методами, могут применяться ключевые слова virtual, override и abstract, что в случае полей является недопустимым. И, наконец, последней особенностью свойств является то, что блоки get и set в них могут иметь собственные уровни доступности, как показано ниже:

```
// Поле, используемое свойством.
private int myInt;
// Свойство.
public int MyIntProp
{
  get
  {
    return myInt;
  }
```

```
    protected set
    {
      myInt = value;
    }
  }
```

Здесь использовать блок `set` сможет только код, находящийся либо внутри самого класса, либо внутри его производных классов.

Уровни доступности, применяемые для блоков `get` и `set`, зависят от уровня доступности самого свойства: делать их доступнее свойства, к которому они относятся, запрещено. Это означает, что в блоках `get` и `set` для свойств с уровнем доступности `private` не может использоваться вообще никаких модификаторов доступности, а для блоков `get` и `set` свойств с уровнем доступности `public`, наоборот, могут быть указаны все возможные модификаторы. В следующем практическом занятии предоставляется возможность поэкспериментировать с определением и использованием полей, методов и свойств.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование полей, методов и свойств

1. Создайте новое консольное приложение по имени `Ch10Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter10`.
2. Добавьте в него новый класс по имени `MyClass` с помощью ссылки Add Class (Добавить класс), что приведет к определению этого нового класса в отдельном новом файле `MyClass.cs`.
3. Измените код в этом файле `MyClass.cs` следующим образом:

```
public class MyClass
{
  public readonly string Name;
  private int intVal;
  public int Val
  {
    get
    {
      return intVal;
    }
    set
    {
      if (value >= 0 && value <= 10)
       intVal = value;
      else
       throw (new ArgumentOutOfRangeException("Val", value,
         "Val must be assigned a value between 0 and 10."));
         // Val может присваиваться только значение в диапазоне от 0 до 10
      }
    }
    public override string ToString()
    {
      return "Name: " + Name + "\nVal: " + Val;
    }
    private MyClass() : this("Default Name")
    {
    }
    public MyClass(string newName)
    {
      Name = newName;
      intVal = 0;
    }
}
```

Фрагмент кода ***Ch10Ex01\MyClass.cs***

4. Далее измените код в файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("Creating object myObj...");
                  // Создание объекта myObj...
  MyClass myObj = new MyClass("My Object");
  Console.WriteLine("myObj created.");
                  // Объект myObj создан
  for (int i = -1; i <= 0; i++)
  {
    try
    {
      Console.WriteLine("\nAttempting to assign {0} to myObj.Val...", i);
                      // Попытка присвоить myObj.Val значение {0}
      myObj.Val = i;
      Console.WriteLine("Value {0} assigned to myObj.Val.", myObj.Val);
                      // Значение {0} присвоено myObj.Val
    }
    catch (Exception e)
    {
      Console.WriteLine("Exception {0} thrown.", e.GetType().FullName);
                      // Сгенерировано исключение{0}
      Console.WriteLine("Message:\n\"{0}\"", e.Message);
                      // Сообщение: {0}
    }
  }
  Console.WriteLine("\nOutputting myObj.ToString()...");
                    // Вывод myObj.ToString()...
  Console.WriteLine(myObj.ToString());
  Console.WriteLine("myObj.ToString() Output.");
  Console.ReadKey();
}
```

Фрагмент кода ***Ch10Ex01\Program.cs***

5. Запустите приложение. На рис. 10.1 показан результат, который должен получиться.

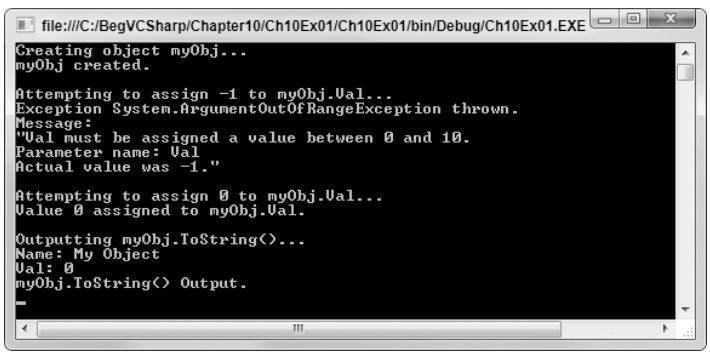


Рис. 10.1. *Приложение* `Ch10Ex01` *в действии*

Описание работы

В коде внутри `Main()` создается и используется экземпляр класса `MyClass`, который был определен в файле `MyClass.cs`. Создание экземпляра этого класса должно осуществляться с применением конструктора, отличного от предлагаемого по умолчанию, потому что конструктор по умолчанию `MyClass` является приватным:

```
private MyClass() : this("Default Name")
{
}
```

Использование конструкции `this("Default Name")` гарантирует получение значения полем `Name` даже в случае вызова этого конструктора, что возможно при использовании данного класса для порождения нового класса. Это необходимо, т.к. не присваивание полю `Name` значения чревато возникновением ошибок в будущем.

Конструктор не по умолчанию предусматривает присваивание значений `readonly`-полю `Name` (это можно делать только в объявлении поля или в конструкторе) и `private`-полю `intVal`.

Далее в методе `Main()` предпринимается две попытки присвоить значения свойству `Val` объекта `myObj` (который является экземпляром класса `MyCalss`). Цикл `for` применяется для присваивания значений `-1` и `0` за два прохода, а структура `try...catch` служит для проверки на предмет возникновения исключения. При присваивании свойству значения `-1` генерируется исключение типа `System.ArgumentOutOfRangeException`, и код в блоке `catch` выводит информацию об этом исключении в окне консоли. В следующем цикле свойству `Val` успешно присваивается значение `0`, которое далее через это свойство присваивается полю `intVal`.

И, наконец, используется переопределенная версия метода `ToString()` для вывода сформатированной строки, отражающей содержимое объекта:

```
public override string ToString()
{
  return "Name: " + Name + "\nVal: " + Val;
}
```

Этот метод должен быть объявлен с ключевым словом `override`, поскольку он переопределяет виртуальный метод `ToString`, определенный в базовом классе `System.Object`. В приведенном здесь коде используется непосредственно свойство `Val`, а не приватное поле `intVal`. Особых причин, по которым бы не следовало подобным образом работать со свойствами в коде классов, не существует, хотя это может немного сказаться на производительности (скорее всего, это вряд ли удастся заметить). Естественно, применение свойства обеспечивает также и проверку достоверности, заложенную в самих свойствах, что может быть выгодно для кода внутри класса.

Добавление членов из диаграммы классов

В предыдущей главе было показано, как применять диаграмму классов для изучения имеющихся в проекте классов. Также там упоминалось, что диаграмму классов можно применять и для добавления членов, и именно об этом пойдет речь в настоящем разделе.

Диаграмма классов представляет собой компонент, который доступен только в VS; в VCE она не поддерживается.

Все инструменты, которые можно применять для добавления и редактирования членов, отображаются в представлении Class Diagram (Диаграмма классов) внутри окна Class Details (Информация о классе). Чтобы увидеть, как это выглядит, давайте создадим диаграмму для класса `MyClass` из приложения `Ch10Ex01` и отобразим список существующих в нем членов, развернув его представление в визуальном конструкторе клас-

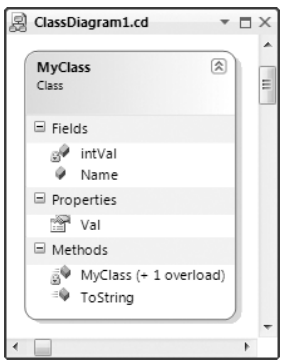


Рис. 10.2. *Члены класса* `MyClass`

сов (щелчком на значке с изображением двух указывающих вниз стрелок). Результирующее представление можно видеть на рис. 10.2.

На рис. 10.3 показано, какая информация появится в окне Class Details после выбора этого класса.

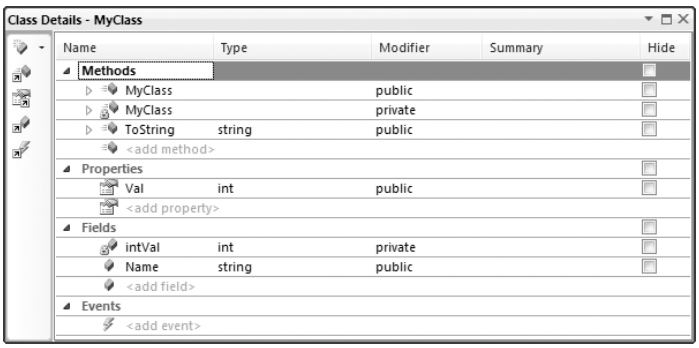


Рис. 10.3. *Просмотр информации о классе* `MyClass` *в окне* Class Details

На этом рисунке видно, что в окне Class Details отображаются все члены, определенные в `MyClass`, а также предлагаются пустые ячейки для добавления новых членов за счет простого ввода их имен.

Добавление методов

Чтобы добавить в класс новый метод, необходимо просто ввести его имя в ячейке с меткой `<add method>` (добавить метод). После ввода имени метода можно нажать клавишу <Tab> и перейти к настройке остальных параметров, начиная с возвращаемого типа, уровня доступности и суммарной информации (которая впоследствии превращается в XML-документацию) и заканчивая указанием того, должен ли метод быть скрыт в диаграмме классов.

Добавив метод, можно развернуть представляющий его узел и аналогичным образом добавить к нему параметры. Для параметров также поддерживается возможность использования модификаторов `out`, `ref` и `params`. На рис. 10.4 показан пример добавления нового метода.

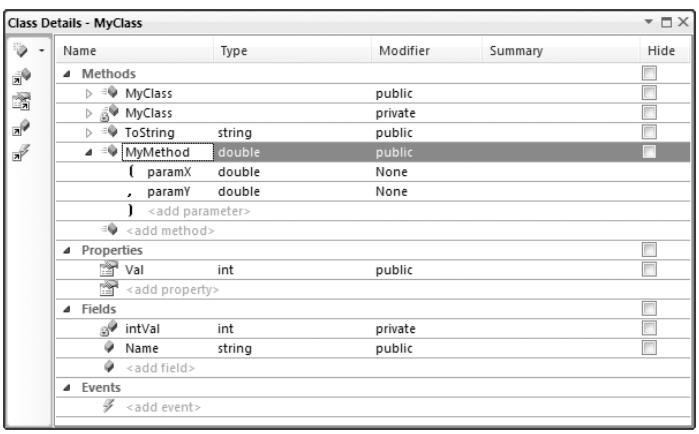


Рис. 10.4. *Добавление нового метода в окне* Class Details

Добавление нового метода, как показано на рис. 10.4, приведет к появлению в классе следующего кода:

```
public double MyMethod(double paramX, double paramY)
{
  throw new System.NotImplementedException();
}
```

Остальные параметры метода можно конфигурировать в окне Properties (Свойства), которое показано на рис. 10.5.

Рис. 10.5. *Окно* Properties *для конфигурирования нового метода*

Помимо всего прочего, здесь метод можно сделать статическим. Очевидно, что автоматически получить реализацию метода такой подход не позволяет, но базовую структуру все-таки предоставляет, а также сокращает количество возможных опечаток.

Добавление свойств

Добавление свойств осуществляется во многом похожим образом. На рис. 10.6 показан пример добавления нового свойства с помощью окна Class Details.

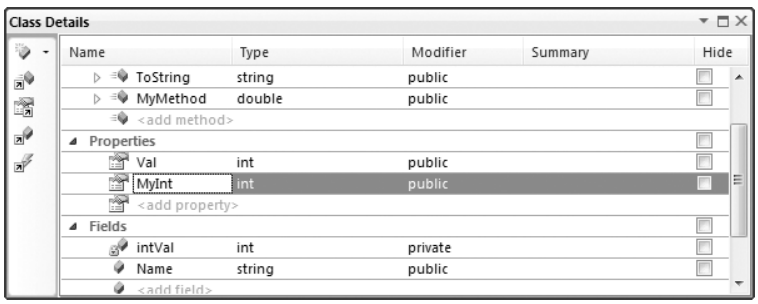


Рис. 10.6. *Добавление нового свойства в окне* Class Details

Добавление такого свойства приведет к добавлению в класс следующего кода:

```
public int MyInt
{
  get
  {
    throw new System.NotImplementedException();
  }
  set {}
}
```

Код реализации понадобится писать самостоятельно, что предполагает сопоставление свойства с полем для простых свойств, удаление блока `set` или `get` для получения свойства только для чтения или только для записи и применение к средствам доступа модификаторов доступности. Тем не менее, базовая структура предоставляется автоматически.

Добавление полей

Добавление полей выполняется точно так же легко. Достаточно просто ввести имя поля, выбрать модификатор типа и доступа.

Рефакторинг членов

Одним из приемов, который может оказаться полезным при добавлении свойств, является возможность генерировать свойство из поля. Это пример *рефакторинга* (refactoring), под которым понимается изменение кода не вручную, а с помощью определенного инструмента. Доступ к инструментам рефакторинга производится по щелчку правой кнопкой на интересующем члене в диаграмме классов или в представлении кода.

В VCE возможности рефакторинга ограничены и, к сожалению, не включают в себя описанный здесь инструмент инкапсуляции поля. В VS доступно гораздо больше средств рефакторинга.

Например, предположим, что в классе `MyClass` имеется следующее поле:

```
public string myString;
```

Щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Refactor⇨Encapsulate Field (Рефакторинг⇨Инкапсулировать поле). Откроется диалоговое окно Encapsulate Field (Инкапсулировать поле), показанное на рис. 10.7.

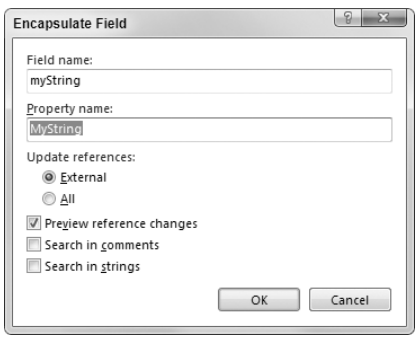


Рис. 10.7. *Диалоговое окно* Encapsulate Field

Если принять предлагаемые по умолчанию настройки в этом окне, код в классе `MyClass` изменится следующим образом:

```
private string myString;
public string MyString
{
  get
  {
    return myString;
  }
  set
  {
    myString = value;
  }
}
```

Здесь уровень доступности поля `myString` был изменен на `private`, а общедоступное свойство с именем `MyString` создано и автоматически связано с полем `myString`. Очевидно, что сокращение времени, требуемого на монотонное создание свойств для полей, является большим плюсом.

Автоматические свойства

Свойства являются предпочтительным способом для получения доступа к состоянию объекта, поскольку они не позволяют никакому внешнему коду реализовать хранилище данных внутри объекта. Вдобавок они обеспечивают больший контроль над способом получения доступа к внутренним данным, как уже несколько раз демонстрировалось в приводившемся выше коде. Обычно, однако, они определяются очень стандартным образом, т.е. за счет создания какого-то приватного члена, доступ к которому должен осуществляться именно через общедоступное свойство. Необходимый для этого код выглядит практически так же, как и представленный в предыдущем разделе, и был автоматически сгенерирован инструментом рефакторинга VS.

Рефакторинг несомненно ускоряет дело в том, что касается ввода кода, но в C# имеется еще одно полезное средство — *автоматические свойства*. Автоматическое свойство объявляется с применением упрощенного синтаксиса, а компилятор C# самостоятельно заполняет все пробелы. В частности, компилятор добавляет объявление приватного (`private`) поля, которое должно применяется для хранения, и использует его в блоках `get` и `set` внутри свойства, не заставляя программиста беспокоиться об этих деталях.

Для определения автоматического свойства используется следующая кодовая структура:

```
public int MyIntProp
{
  get;
  set;
}
```

Можно также определять автоматическое свойство в одной строке кода для экономии места:

```
public int MyIntProp { get; set; }
```

Уровень доступности, тип и имя свойства определяются обычным образом, но код реализации блоков `get` и `set` не предоставляется. Код реализации этих блоков (и лежащего в основе поля) генерируются самим компилятором.

При использовании автоматического свойства доступ к его данным можно получать только через это свойство, но не через лежащее в основе приватное поле, поскольку нельзя получить доступ к лежащему в основе приватному полю, не зная его имени, которое станет известным лишь во время компиляции. Однако это нельзя расценивать как ограничение, так как вполне достаточно имени свойства. Единственным настоящим ограничением автоматических свойств является то, что они обязательно должны включать в себя и блок `get`, и блок `set`, т.е. определять с их помощью свойства, доступные только для чтения или только для записи, нельзя.

Дополнительные темы, связанные с членами классов

Теперь можно переходить к рассмотрению более сложных тем, связанных с членами. В настоящем разделе речь пойдет о следующих моментах.

- Сокрытие методов базового класса.
- Вызов переопределенных или скрытых методов базового класса.
- Вложенные определения типов.

Сокрытие методов базового класса

При наследовании (не абстрактного) члена базового класса также наследуется и его реализация. Когда наследуемый член является виртуальным, его реализацию допускается переопределить с помощью ключевого слова `override`. Вдобавок, независимо от того, является член виртуальным или нет, реализацию при желании можно *сокрыть*. Это полезно в ситуации, например, когда общедоступный наследуемый член работает не совсем так, как нужно.

Сокрытие осуществляется с применением показанного ниже кода:

```
public class MyBaseClass
{
  public void DoSomething()
  {
    // Базовая реализация.
  }
}
public class MyDerivedClass : MyBaseClass
{
  public void DoSomething()
  {
    // Реализация в производном классе, скрывающая базовую реализацию.
  }
}
```

Несмотря на то что этот код работает нормально, он приводит к выдаче предупреждающего сообщения о сокрытии члена базового класса. Это позволяет исправить ситуацию, если член был сокрыт по ошибке и на самом деле должен использоваться. Если же действительно требуется сокрыть член класса, можно воспользоваться ключевым словом `new` для явного указания этой операции:

```
public class MyDerivedClass : MyBaseClass
{
  new public void DoSomething()
  {
    // Реализация в производном классе, скрывающая базовую реализацию.
  }
}
```

Приведенный код работает точно так же, но не приводит к выводу каких-либо предупреждений. А теперь не помешает разобраться с тем, а чем сокрытие отличается от переопределения членов базового класса. Для примера рассмотрим следующий код:

```
public class MyBaseClass
{
  public virtual void DoSomething()
  {
    Console.WriteLine("Base imp");    // Базовая реализация
  }
}
```

```
public class MyDerivedClass : MyBaseClass
{
  public override void DoSomething()
  {
    Console.WriteLine("Derived imp");
                    // Производная реализация

  }
}
```

Здесь реализация метода в базовом классе заменяется его переопределенной версией, из-за чего в следующем коде будет использоваться уже новая версия, хотя и все равно через тип базового класса (за счет полиморфизма):

```
MyDerivedClass myObj = new MyDerivedClass();
MyBaseClass myBaseObj;
myBaseObj = myObj;
myBaseObj.DoSomething();
```

Результатом выполнения этого кода будет следующая строка:

```
Derived imp
```

В качестве альтернативы можно было бы просто сокрыть метод базового класса:

```
public class MyBaseClass
{
  public virtual void DoSomething()
  {
    Console.WriteLine("Base imp");
                    // Базовая реализация
  }
}
public class MyDerivedClass : MyBaseClass
{
  new public void DoSomething()
  {
    Console.WriteLine("Derived imp");
                    // Производная реализация
  }
}
```

Чтобы этот код работал, метод базового класса необязательно должен быть виртуальным. Эффект получается точно таким же, и по сравнению с предыдущим кодом изменить требуется всего лишь одну строку. Результат выполнения этого кода, как в случае виртуального, так и в случае не виртуального метода базового класса, будет выглядеть так:

```
Base imp
```

Хотя базовая реализация сокрыта, к ней по-прежнему можно получать доступ через базовый класс.

Вызов переопределенных или скрытых методов базового класса

Как при переопределении, так и при сокрытии доступ к члену базового класса из производного класса все равно возможен. Ситуаций, в которых это может оказаться полезным, существует довольно много, например:

- нужно сокрыть какой-то унаследованный общедоступный член от пользователей производного класса, но при этом все равно иметь возможность получать доступ к его функциональности внутри данного класса;
- необходимо добавить реализацию унаследованного виртуального класса вместо того, чтобы просто заменять ее новой переопределенной реализацией.

Для этого служит ключевое слово base, которое позволяет ссылаться на реализацию базового класса внутри производного (подобно тому, как оно применялось для управления конструкторами в предыдущей главе):

```
public class MyBaseClass
{
  public virtual void DoSomething()
  {
    // Базовая реализация.
  }
}
public class MyDerivedClass : MyBaseClass
{
  public override void DoSomething()
  {
    // Реализация в производном классе, расширяющая базовую реализацию.
    base.DoSomething();
    // Дополнительная реализация в производном классе.
  }
}
```

В этом коде предусмотрено выполнение версии DoSomething, которая содержится в MyBaseClass, являющимся базовым классом для MyDerivedClass, внутри версии DoSomething, которая содержится в MyDerivedClass. Поскольку ключевое слово base работает с экземплярами объектов, использование его в статическом члене приведет к ошибке.

Ключевое слово this

Вдобавок к base можно также использовать ключевое слово this. Как и base, ключевое слово this может быть указано в членах класса и, подобно base, оно позволяет ссылаться на экземпляр объекта, но только на текущий (т.е. применять его в статических членах нельзя, потому что статические члены не являются частью экземпляра объекта).

Наиболее полезной функцией ключевого слова this является его способность передавать методу ссылку на текущий экземпляр объекта, как показано в следующем примере:

```
public void doSomething()
{
  MyTargetClass myObj = new MyTargetClass();
  myObj.DoSomethingWith(this);
}
```

Здесь создаваемый экземпляр MyTargetClass (по имени myObj) имеет метод под названием DoSomethingWith(), который принимает единственный параметр типа, совместимого с типом класса, содержащего предыдущий метод. Типом этого параметра может быть тип данного класса, тип класса, производным от которого является данный класс, тип интерфейса, реализуемого данным классом, либо (конечно же) тип System.Object.

Также ключевое слово this применяется для уточнения членов локального типа, например:

```
public class MyClass
{
  private int someData;
  public int SomeData
  {
    get
    {
      return this.someData;
    }
  }
}
```

Многим разработчиками нравится такой синтаксис, поскольку его можно использовать с членами любого типа и по нему сразу понятно, что имеется в виду член, а не локальная переменная.

Вложенные определения типов

Типы наподобие классов можно определять не только в пространствах имен, но и внутри других классов. Тогда для них можно использовать весь спектр модификаторов доступности, а не только модификаторы `public` и `internal`, а также ключевое слово `new` для сокрытия тех из них, которые наследуются от какого-то базового класса. Например, в следующем коде помимо класса `MyClass` определяется вложенный класс по имени `myNestedClass`:

```
public class MyClass
{
  public class myNestedClass
  {
    public int nestedClassField;
  }
}
```

Для создания экземпляра `myNestedClass` за пределами `myClass` его имя должно обязательно уточняться, как показано ниже:

```
MyClass.myNestedClass myObj = new MyClass.myNestedClass();
```

Однако подобное может оказаться совершенно невозможным, если вложенный класс объявлен с уровнем доступности `private` или каким-то другим уровнем, являющимся несовместимым с кодом, который выполняется на этапе создания такого экземпляра. Возможность создавать вложенные определения типов главным образом предусмотрена для того, чтобы позволить определять классы, являющиеся приватными по отношению к содержащему их классу; никакой другой код в пространстве имен не сможет получать к ним доступ.

Реализация интерфейсов

Прежде чем двигаться дальше, давайте подробнее посмотрим, как определяются и реализуются интерфейсы. В предыдущей главе рассказывалось о том, что интерфейсы определяются подобно классам с применения примерно такого кода:

```
interface IMyInterface
{
  // Члены интерфейса.
}
```

Члены интерфейсов тоже определяются подобно членам классов, но имеют несколько важных отличий, которые перечислены ниже.

- При определении членов интерфейсов не разрешено применение каких-либо модификаторов доступа (`public`, `private`, `protected` или `internal`); все члены интерфейсов являются неявно общедоступными.
- Члены интерфейсов не могут содержать код.
- В интерфейсах не могут присутствовать члены типа полей.
- Члены интерфейсов не могут определяться с использованием таких ключевых слов, как `static`, `virtual`, `abstract` или `sealed`.
- В члены интерфейсов запрещено добавлять определения типов.

Однако можно определять члены интерфейсов с использованием ключевого слова new, чтобы сокрыть члены, унаследованные от базовых интерфейсов:

```
interface IMyBaseInterface
{
  void DoSomething();
}
interface IMyDerivedInterface : IMyBaseInterface
{
  new void DoSomething();
}
```

Это работает в точности так же, как и сокрытие унаследованных членов класса.

В свойства, определяемые в интерфейсах, можно включать как какой-то один, так и оба блока get и set в зависимости от того, какой уровень доступа требуется к ним предоставить:

```
interface IMyInterface
{
  int MyInt { get; set; }
}
```

Здесь int-свойство MyInt включает оба блока, get и set. В случае создания свойства с более ограниченным доступом какой-то из них может быть опущен.

Такой синтаксис напоминает синтаксис автоматических свойств, но важно помнить о том, что автоматические свойства могут определяться только для классов, но не для интерфейсов, и что в них обязательно должны присутствовать оба блока, get и set.

То, как должны храниться данные свойств, в интерфейсах не специфицируется. Например, в них нельзя указывать поля, которые могут использоваться для хранения данных свойств. И, наконец, интерфейсы, как и классы, могут определяться в виде членов классов (но не в виде членов других экземпляров, поскольку содержать определения типов им не разрешено).

Реализация интерфейсов в классах

В классе, который реализует интерфейс, должны обязательно присутствовать реализации всех членов этого интерфейса. Эти реализации должны соответствовать сигнатурам, которые были указаны (в том числе заданным блокам get и set), и быть общедоступными:

```
public interface IMyInterface
{
  void DoSomething();
  void DoSomethingElse();
}
public class MyClass : IMyInterface
{
  public void DoSomething()
  {
  }
  public void DoSomethingElse()
  {
  }
}
```

Члены интерфейсов допускается реализовать с использованием ключевого слова virtual или abstract, но не ключевого слова static или constant. Кроме того, они могут быть реализованы в базовых классах:

```
public interface IMyInterface
{
  void DoSomething();
  void DoSomethingElse();
}
public class MyBaseClass
{
  public void DoSomething()
  {
  }
}
public class MyDerivedClass : MyBaseClass, IMyInterface
{
  public void DoSomethingElse()
  {
  }
}
```

Наследование от базового класса, в котором реализуется заданный интерфейс, означает, что этот интерфейс будет неявно поддерживаться и в производном классе. Например:

```
public interface IMyInterface
{
  void DoSomething();
  void DoSomethingElse();
}
public class MyBaseClass : IMyInterface
{
  public virtual void DoSomething()
  {
  }
  public virtual void DoSomethingElse()
  {
  }
}
public class MyDerivedClass : MyBaseClass
{
  public override void DoSomething()
  {
  }
}
```

Очевидно, что в базовых классах реализации лучше определять как виртуальные, чтобы потом иметь возможность заменять, а не скрывать их в производных классах. Если скрыть член базового класса с помощью ключевого слова `new`, а не переопределить его, как было показано здесь, метод `IMyInterface.DoSomething` всегда бы ссылался на версию базового класса, даже если бы доступ к производному классу осуществлялся через интерфейс.

Явная реализация членов интерфейса

Члены интерфейсов также могут быть реализованы *явно* классом. В случае применения такого подхода доступ к членам возможен только через интерфейс, но не класс. Доступ к *неявным* членам, которые использовались в коде из предыдущего раздела, можно осуществлять обоими способами.

Например, в случае неявной реализации в классе `MyClass` метода `DoSomething()` из интерфейса `IMyInterface`, как в предыдущем примере, следующий код будет вполне допустим:

```
MyClass myObj = new MyClass();
myObj.DoSomething();
```

И этот код также будет правильным:

```
MyClass myObj = new MyClass();
IMyInterface myInt = myObj;
myInt.DoSomething();
```

При явной реализации в MyDerivedClass метода DoSomething() допустимым будет только второй подход. Необходимый код показан ниже:

```
public class MyClass : IMyInterface
{
  void IMyInterface.DoSomething()
  {
  }
  public void DoSomethingElse()
  {
  }
}
```

Здесь метод DoSomething() реализуется явно, а метод DoSomethingElse() — неявно. Доступ непосредственно через экземпляр объекта myClass возможен только к DoSomethingElse().

Добавление средств доступа, отличных от public

Ранее было заявлено, что в случае реализации интерфейса с каким-нибудь свойством должны быть обязательно реализованы соответствующие блоки get и set. Это не совсем так — допускается также добавлять блок get в свойство внутри класса, где определяющий это свойство интерфейс имеет только блок set, и наоборот. Однако такое возможно только в случае использования для этого блока более ограничивающего модификатора доступа, чем у того, что определен в интерфейсе. Поскольку блоки, определяемые в интерфейсе, изначально имеют уровень доступа public, получается, что добавлять можно только блоки с уровнем, отличным от public. Ниже приведен пример:

```
public interface IMyInterface
{
  int MyIntProperty
  {
    get;
  }
}
public class MyBaseClass : IMyInterface
{
   public int MyIntProperty { get; protected set; }
}
```

Частичные определения классов

При создании классов с множеством членов того или иного типа дело может довольно сильно усложняться, а файлы кода — становится очень длинными. Одним из приемов, который помогает в подобных случаях, и который уже демонстрировался в предыдущих главах, является организации кода по разделам. Разделы в коде впоследствии могут сворачиваться и разворачиваться, улучшая восприятие кода. Например, определение класса может выглядеть следующим образом:

```
public class MyClass
{
  #region Fields
  private int myInt;
  #endregion
```

```
  #region Constructor
  public MyClass()
  {
    myInt = 99;
  }
  #endregion

  #region Properties
  public int MyInt
  {
    get
    {
      return myInt;
    }
    set
    {
      myInt = value;
    }
  }
  #endregion

  #region Methods
  public void DoSomething()
  {
    // Выполнение каких-нибудь действий...
  }
  #endregion
}
```

В результате появляется возможность разворачивать и сворачивать поля, свойства, конструктор и методы класса и, следовательно, концентрировать внимание только на том коде, который представляет интерес в текущий момент. Можно также создавать вложенные разделы.

Тем не менее, даже в случае применения такого приема ситуация все равно может выходить из-под контроля. Существует еще одна альтернатива — использование *частичных определений классов*. Попросту говоря, этот прием предусматривает разнесение определения класса по нескольким файлам, с помещением, например, определения полей, свойств и конструктора в один файл, а методов — в другой. Все, что при этом требуется — указывать ключевое слово `partial` возле имени класса в каждом из файлов, где размещается та или иная часть его определения:

```
public partial class MyClass
{
...
}
```

В случае применения частичных определений классов ключевое слово `partial` должно обязательно присутствовать в каждом из содержащих часть определения файлов.

Частичные классы очень часто и вполне успешно используются в Windows-приложениях для сокрытия кода, отвечающего за компоновку форм. На самом деле мы уже видели нечто подобное в главе 2, а именно — что код формы Windows в классе по имени `Form1`, например, хранится в файлах `Form1.cs` и `Form1.Designer.cs`. Это позволяет сосредоточить внимание на функциональности формы и не беспокоиться по поводу дополнении кода информацией, которая не особо интересна.

Напоследок важно обратить внимание на еще одну касающуюся частичных классов деталь — действие интерфейсов, применяемых к какой-то одной части частичного класса, распространяется на весь этот класс. Это означает, что определение:

```
public partial class MyClass : IMyInteface1
{
  ...
}
public partial class MyClass : IMyInteface2
{
  ...
}
```

будет полностью эквивалентно следующему определению:

```
public class MyClass : IMyInteface1, IMyInteface2
{
  ...
}
```

Базовый класс в частичных определениях классов может быть указан внутри как одного, так и нескольких частичных определений. Однако в случае указания в нескольких местах это должен быть *один и тот же* базовый класс, поскольку, как известно, классы в C# могут наследоваться только от одного базового класса.

Частичные определения методов

В частичных классах можно также определять частичные методы. Для этого в одном частичном определении класса объявляется метод без тела, а реализация его предоставляется в другом частичном определении класса. В обоих местах определение метода должно сопровождаться ключевым словом `partial`:

```
public partial class MyClass
{
  partial void MyPartialMethod();
}
public partial class MyClass
{
  partial void MyPartialMethod()
  {
    // Реализация метода
  }
}
```

Частичные методы также могут быть статическими, хотя они всегда являются приватными и не имеют возвращаемого значения. Использовать в них параметры `out` не допускается, но зато разрешено применять параметры `ref`. С ними нельзя использовать такие модификаторы, как `virtual`, `abstract`, `override`, `new`, `sealed` и `extern`.

Из-за всех этих ограничений сразу может быть не понятно, для чего нужны такие методы. На самом деле частичные методы играют важную роль при компиляции кода, а не при его использовании. Для примера рассмотрим следующий код:

```
public partial class MyClass
{
  partial void DoSomethingElse();

  public void DoSomething()
  {
    Console.WriteLine("DoSomething() execution started.");
                    // Началось выполнение метода DoSomething()
    DoSomethingElse();
    Console.WriteLine("DoSomething() execution finished.");
                    // Выполнение метода DoSomething() завершено
  }
}
```

```
public partial class MyClass
{
  partial void DoSomethingElse()
  {
    Console.WriteLine("DoSomethingElse() called.");
                    // Был вызван метод DoSomethingElse()
  }
}
```

Здесь частичный метод `DoSomethingElse` определяется и вызывается в первом определении частичного класса, а реализуется — во втором. При вызове `DoSomething` из консольного приложения вывод будет выглядеть так, как и ожидалось:

```
DoSomething() execution started.
DoSomethingElse() called.
DoSomething() execution finished.
```

Однако если удалить второе определение частичного класса или всей реализации частичного метода (либо закомментировать этот код), вывод будет выглядеть следующим образом:

```
DoSomething() execution started.
DoSomething() execution finished.
```

Может показаться, что в таком случае происходит следующее: при вызове метода `DoSomethingElse` исполняющая среда обнаруживает, что у этого метода нет реализации, и потому переходит к выполнению следующей строки кода. Но на самом деле предпринимаются более изощренные действия. При компиляции кода, в котором содержится определение частичного метода без реализации, компилятор вообще удаляет весь этот метод, а также любые связанные с ним вызовы. По этой причине во время выполнения кода никакой проверки не предмет наличия реализации не производится, поскольку не существует вызова, который бы привел к такой проверке. В результате будет наблюдаться хоть и небольшое, но все равно важное улучшение производительности.

Как и частичные классы, частичные методы полезны при настройке автоматически генерируемого или создаваемого визуальным конструктором кода. Визуальный конструктор может объявлять частичные методы, а разработчик затем — либо реализовать, либо не реализовать их, в зависимости от ситуации. Если методы не реализованы, производительность не пострадает, т.к. в скомпилированном коде этих методов не существует.

А теперь подумайте, почему частичные методы не могут иметь возвращаемый тип? Читатели, ответившие самостоятельно на этот вопрос, могут считать, что полностью разобрались в данной теме. Теперь рассмотрим практический пример.

Пример приложения

Чтобы увидеть, как некоторые из описанных выше приемов применяются на практике, в настоящем разделе предлагается создать модуль классов, который будет дорабатываться и использоваться в последующих главах книги. В состав этого модуля должно входить два класса.

- Класс `Card`, представляющий стандартную игральную карту трефовой, бубновой, червовой или пиковой масти достоинством от туза до короля.
- Класс `Deck`, представляющий полную колоду из 52 карт, в котором предусмотрена возможность доступа к картам по их позиции в колоде, а также тасования колоды.

Кроме того, будет построено простое клиентское приложение для проверки работоспособности модуля; применение модуля в полнофункциональном карточном приложении пока демонстрироваться не будет.

Продумывание приложения

В библиотеке классов данного приложения, которая будет называться `Ch10CardLib`, должны содержаться классы. Прежде чем приступать к написанию кода, нужно хорошо продумать структуру и функциональные возможности этих классов.

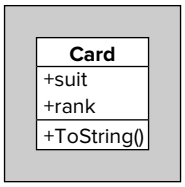


Рис. 10.8. *Класс* `Card`

Класс `Card`

Класс `Card`, по сути, представляет собой контейнер, содержащий два доступных только для чтения поля: `suit` (масть) и `rank` (достоинство). Причина того, что эти поля должны быть доступны только для чтения, состоит в том, что иметь "пустую" карту нет никакого смысла, да и изменять карты после их создания тоже не должно допускаться. Для простоты можно сделать конструктор по умолчанию приватным и предоставить альтернативный конструктор, способный создавать карту указанной масти и достоинства.

Помимо этого, класс `Card` будет переопределять метод `ToString` из класса `SystemObject`, чтобы иметь возможность легко получать представляющую карту строку в понятном для восприятия виде. Чтобы еще немного упростить дело, для полей `suit` и `rank` будут предоставлены соответствующие перечисления.

Класс `Card` показан на рис. 10.8.

Класс `Desk`

Класс `Deck` должен обслуживать 52 объекта `Card`. Для этого в нем можно использовать простой массив. Получать доступ напрямую к этому массиву не нужно, потому что доступ к каждому объекту `Card` должен осуществляться через метод `GetCard`, возвращающий объект `Card` с указанным индексом. Кроме того, в классе `Deck` должен поддерживаться метод `Shuffle` для тасования карт в массиве. Класс `Deck` показан на рис. 10.9.

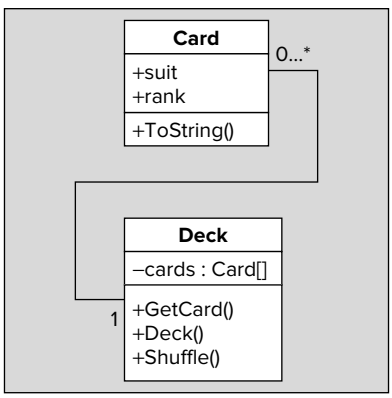


Рис. 10.9. *Класс* `Deck`

Написание библиотеки классов

В этом примере предполагается, что читатель уже достаточно хорошо знаком с IDE-средой, поэтому подробное описание шагов не приводится. Основное внимание уделяется детальному рассмотрению кода, хотя некоторые указания все же даются для исключения вероятности возникновения по ходу каких-нибудь проблем.

Оба класса и перечисления должны содержаться в проекте типа библиотеки классов по имени `Ch10CardLib`. Этот проект будет содержать четыре файла `.cs`: `Card.cs` с опреде-

лением класса `Card`, `Deck.cs` с определением класса `Deck`, а также `Suit.cs` и `Rank.cs` с определениями соответствующих перечислений.

Большую часть необходимого кода можно собрать с помощью предлагаемого в составе VS инструмента для построения диаграмм классов.

Тем, кто использует VCE и потому не имеет доступа к инструменту для построения диаграмм, не стоит беспокоиться. В последующих разделах будет приведен весь генерируемый этим инструментом код, благодаря которому можно отслеживать происходящий процесс.

Для начала выполните перечисленные ниже действия.

1. Создайте новый проект типа библиотеки классов, назначьте ему имя `Ch10CardLib` и сохраните в каталоге `C:\BegVCSharp\Chapter10`.
2. Удалите из этого проекта файл `Class1.cs`.
3. Если используется VS, откройте диаграмму классов для этого проекта в окне Solution Explorer (чтобы появился значок диаграммы классов, должен быть выбран проект, а не решение). Вначале диаграмма классов должна быть пустой, поскольку в проекте пока не содержится никаких классов.

Если в диаграмме видны классы `Resources` и `Settings`, их можно скрыть, щелкнув на каждом из них правой кнопкой мыши и выбрав в контекстном меню пункт Remove from Diagram (Удалить из диаграммы).

Добавление перечислений `Suit` и `Rank`

Чтобы добавить перечисление в диаграмму классов, необходимо перетащить в нее элемент Enum (Перечисление) из панели инструментов (Toolbox) и затем соответствующим образом заполнить поля в открывшемся диалоговом окне New Enum (Новое перечисление). Например, для перечисления `Suit` это окно должно быть заполнено так, как показано на рис. 10.10.

Рис. 10.10. *Добавление перечисления `Enum`*

Теперь необходимо добавить члены перечисления с помощью окна Class Details (Информация о классе). Значения, требуемые для перечисления `Suit`, показаны на рис. 10.11.

Далее следует добавить перечисление `Rank` из панели Toolbox. Значения, необходимые для перечисления `Rank`, показаны на рис. 10.12.

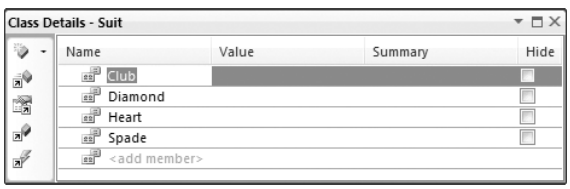


Рис. 10.11*. Добавление членов перечисления `Enum`*

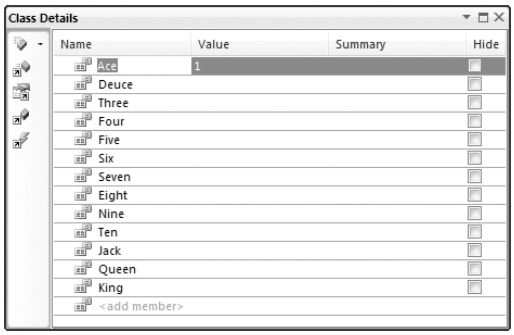


***Рис. 10.12**. Добавление членов перечисления `Rank`*

Значение первого члена, `Ace`, должно быть установлено в `1`, чтобы при сохранении данных в перечислении соблюдалось соответствие с достоинством карты, т.е., например, карта `Six` (шестерка) будет сохранена со значением `6`.

По завершении диаграмма должна приобрести такой вид, как показано на рис. 10.13.

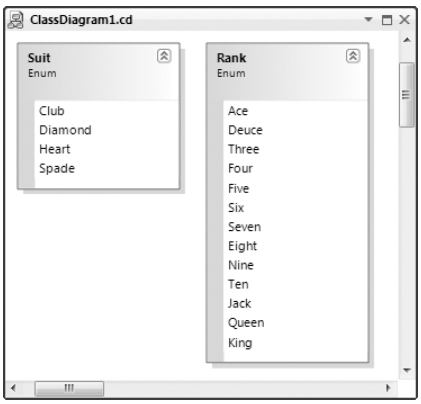


***Рис. 10.13**. Диаграмма классов после добавления перечислений `Suit` и `Rank`*

Код, сгенерированный для этих двух перечислений в файлах `Suit.cs` и `Rank.cs`, будет выглядеть следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch10CardLib
{
  public enum Suit
  {
    Club,
    Diamond,
    Heart,
    Spade,
  }
}
```

*Фрагмент кода **`Ch10CardLib\Suit.cs`***

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch10CardLib
{
  public enum Rank
  {
    Ace = 1,
    Deuce,
    Three,
    Four,
    Five,
    Six,
    Seven,
    Eight,
    Nine,
    Ten,
    Jack,
    Queen,
    King,
  }
}
```

Фрагмент кода ***Ch10CardLib\Rank.cs***

Если вы используете VCE, добавьте сначала файлы `Suit.cs` и `Rank.cs`, а затем введите приведенный выше код. В этом коде обратите внимание на дополнительные запятые, добавленные генератором кода после последнего члена перечисления. Они не препятствуют компиляции и не приводят к созданию дополнительного "пустого" члена (хотя выглядят немного неаккуратно).

Добавление класса `Card`

Для добавления класса `Card` используется визуальный конструктор классов и редактор кода в VS либо только редактор кода в VCE. Добавление класса в конструкторе классов осуществляется во многом так же, как и добавление перечисления, а именно — за счет перетаскивания в диаграмму соответствующего элемента из панели Toolbox. В данном случае требуется перетащить в диаграмму элемент Class и назначить ему имя `Card`.

Далее в окне Class Details необходимо добавить поля `rank` и `suit` и затем установить в окне Properties для свойства `Constant Kind` каждого из этих полей значение `readonly`. Также нужно добавить два конструктора: конструктор по умолчанию (с уровнем доступности `private`) и конструктор, принимающий два параметра — `newSuit` и `newRank` типа `Suit` и `Rank` (с уровнем доступности `public`). И, наконец, понадобится переопределить метод `ToSTring`, для чего следует изменить значение свойства `Inheritance Modifier` в окне Properties на `override`.

На рис. 10.14 показано, как должно выглядеть окно Class Details после ввода всей информации, необходимой для класса `Card`.

Теперь необходимо изменить код класса `Card` в файле `Card.cs`, как показано ниже (или добавить этот код в новый класс по имени `Card` в пространстве имен `Ch10CardLib`, если пример выполняется в среде VCE):

```
public class Card
{
  public readonly Suit suit;
  public readonly Rank rank;
```

```
  public Card(Suit newSuit, Rank newRank)
  {
    suit = newSuit;
    rank = newRank;
  }
  private Card()
  {
  }
  public override string ToString()
  {
    return "The " + rank + " of " + suit + "s";
  }
}
```

Фрагмент кода ***Ch10CardLib\Card.cs***

Переопределенный метод `ToString` возвращает строковое представление хранящегося в перечислении значения, а конструктор не по умолчанию инициализирует значения полей `suit` и `rank`.

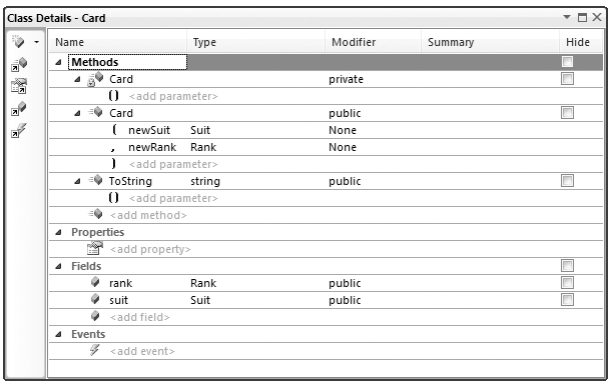


Рис. 10.14*. Окно* Class Details *после ввода информации для класса* `Card`

Добавление класса `Deck`

В класс `Deck` потребуется добавить следующие члены с использованием диаграммы классов.

- Приватное поле с именем `cards` и типом `Card[]`.
- Общедоступный конструктор по умолчанию.
- Общедоступный метод по имени `GetCard()`, принимающий один параметр `cardNumb` типа `int` и возвращающий объект типа `Card`.
- Общедоступный метод по имени `Shuffle()`, не принимающий параметров и возвращающий `void`.

После добавления этих членов информация в окне Class Details для класса `Deck` должна выглядеть так, как показано на рис. 10.15.

Чтобы еще больше прояснить картину на диаграмме, можно отобразить отношения между добавленными членами и типами. Для этого необходимо по очереди щелкнуть правой кнопкой мыши на каждом из членов в диаграмме классов и выбрать в контекстном меню пункт Show as Association (Показать как ассоциацию):

- `cards` в `Deck`
- `suit` в `Card`
- `rank` в `Card`

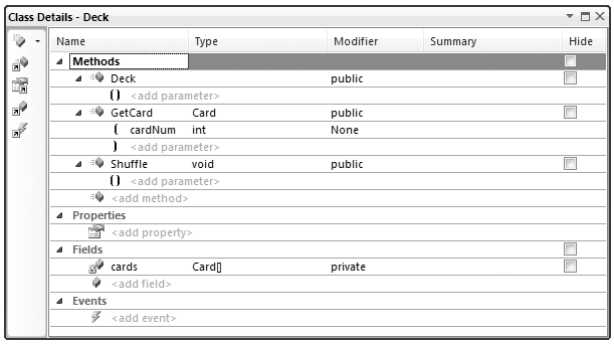


***Рис. 10.15**. Окно* Class Details *после ввода информации для класса* `Deck`

После этого диаграмма должна приобрести такой вид, как на рис. 10.16.

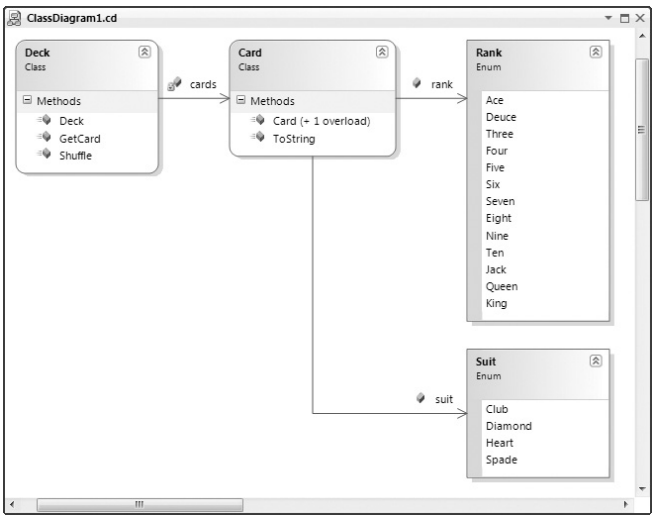


***Рис. 10.16**. Диаграмма классов после отображения отношений*

Теперь нужно модифицировать код в файле `Deck.cs` образом, как показано ниже (в случае использования VCE потребуется добавить новый класс и ввести в нем приведенный здесь код). В коде первым делом реализуется конструктор, в котором просто создаются и присваиваются 52 карты в поле `cards`. Производится итерация по всем комбинациям значений двух перечислений, при этом каждая комбинация используется для создания карты. В результате в `cards` помещается упорядоченный список карт.

```
using System;
using System.Collections.Generic;
using System.Linq
using System.Text;

namespace Ch10CardLib
{
  public class Deck
  {
    private Card[] cards;
```

```
  public Deck()
  {
    cards = new Card[52];
    for (int suitVal = 0; suitVal < 4; suitVal++)
    {
      for (int rankVal = 1; rankVal < 14; rankVal++)
      {
        cards[suitVal * 13 + rankVal -1] = new Card((Suit)suitVal,
                                                    (Rank)rankVal);
      }
    }
  }
```

Фрагмент кода ***Ch10CardLib\Deck.cs***

После этого необходимо реализовать метод `GetCard()`, который либо возвращает объект `Card` по запрошенному индексу, либо генерирует исключение, как было показано ранее:

```
public Card GetCard(int cardNum)
{
  if (cardNum >= 0 && cardNum <= 51)
    return cards[cardNum];
  else
      throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
             "Value must be between 0 and 51."));
}
```

И, наконец, понадобится реализовать метод `Shuffle()`, который создает временный массив карт и копирует в него карты из существующего массива `cards` случайным образом. Большую часть тела этого метода занимает цикл, выполняющий итерации от 0 до 51. На каждой итерации цикла производится генерация случайного числа от 0 до 51 с использованием класса `System.Random`, доступного в .NET Framework. После создания объект этого класса генерирует случайное число от `0` до `X` с помощью метода `Next(X)`. Полученное случайное число служит в качестве индекса во временном массиве, по которому будет копироваться объект `Card` из массива `cards`.

Для фиксирования информации об уже присвоенных картах предусмотрен массив значений `bool`, которым по мере копирования карт присваивается `true`. Во время генерации случайных чисел производится сверка с этим массивом для выяснения того, не была ли скопирована какая-то карта в место во временном массиве, на которое указывает сгенерированное случайное число. Если была, тогда просто генерируется очередное случайное число.

Особо эффективным такой подход назвать нельзя, поскольку до обнаружения свободного места для копирования карты может генерироваться немало случайных чисел. Однако он работает, является очень простым и позволяет коду на C# выполняться настолько быстро, что задержки вряд ли удастся заметить. Ниже приведен полный код метода `Shuffle`:

```
  public void Shuffle()
  {
    Card[] newDeck = new Card[52];
    bool[] assigned = new bool[52];
    Random sourceGen = new Random();
    for (int i = 0; i < 52; i++)
    {
      int destCard = 0;
      bool foundCard = false;
      while (foundCard == false)
      {
        destCard = sourceGen.Next(52);
        if (assigned[destCard] == false)
          foundCard = true;
      }
```

```
        assigned[destCard] = true;
        newDeck[destCard] = cards[i];
      }
      newDeck.CopyTo(cards, 0);
    }
  }
}
```

В последней строке кода этого метода используется метод `CopyTo` класса `System.Array` (который применяется всякий раз, когда создается массив) для копирования карты из `newDeck` в `cards`. Это означает, что вместо создания новых экземпляров используется один и тот же набор объектов `Card` из одного и того же объекта `cards`. Если использовать вместо этого `cards = newDeck`, то экземпляр объекта, на который ссылается `cards`, будет заменен другим, что приведет к появлению проблем при наличии еще где-нибудь в коде ссылки на исходный экземпляр `cards`, который бы так и остался не перетасованным.

На этом создание кода библиотеки классов завершено.

Клиентское приложение для библиотеки классов

Чтобы ничего не усложнять, клиентское консольное приложение можно добавить прямо в то же решение, в котором содержится уже готовая библиотека классов. Для этого щелкните правой кнопкой мыши на этом решении в окне Solution Explorer и выберите в контекстном меню пункт Add⇨New Project (Добавить⇨Новый проект), после чего назначьте новому проекту имя `Ch10CardClient`.

Для использования созданной ранее библиотеки классов в новом проекте консольного приложения сначала необходимо добавить в него ссылку на проект этой библиотеки классов по имени `Ch10CardLib`. Это делается на вкладке Projects (Проекты) диалогового окна Add Reference (Добавление ссылки), как показано на рис. 10.17.

Просто выберите проект и щелкните на кнопке OK; ссылка будет добавлена.

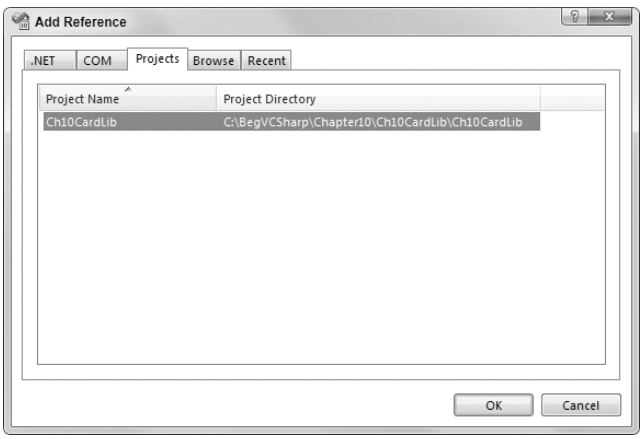


***Рис. 10.17**. Добавление ссылки на проект библиотеки классов*

Поскольку этот новый проект является вторым создаваемым проектом, необходимо указать, что он должен исполнять роль стартового проекта в решении, т.е. запускаться по щелчку на кнопке Run (Выполнить). Для этого щелкните правой кнопкой мыши на имени этого проекта в окне Solution Explorer и выберите в контекстном меню пункт Set as StartUp Project (Сделать стартовым проектом).

Далее добавьте код, в котором используются созданные ранее классы. Ничего особенного в нем делать не требуется, поэтому следующий код вполне подойдет:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch10CardLib;
namespace Ch10CardClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Deck myDeck = new Deck();
      myDeck.Shuffle();
      for (int i = 0; i < 52; i++)
      {
        Card tempCard = myDeck.GetCard(i);
        Console.Write(tempCard.ToString());
        if (i != 51)
          Console.Write(", ");
        else
          Console.WriteLine();
      }
      Console.ReadKey();
    }
  }
}
```

Фрагмент кода ***Ch10CardClient\Program.cs***

На рис. 10.18 показан результат выполнения этого кода.

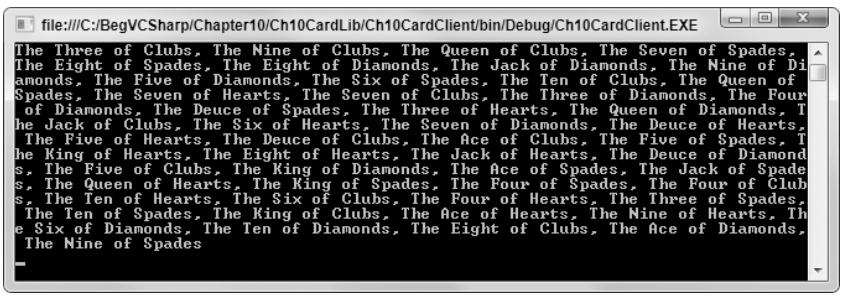


Рис. 10.18*. Клиентское консольное приложение* `Ch10CardClient` *в действии*

Как видно на рисунке, результатом будет тасование 52 игральных карт в колоде. В последующих главах будет продолжена разработка и использование данной библиотеки классов.

Окно Call Hierarchy

Теперь можно кратко рассмотреть новое средство, появившееся в версии VS 2010: окно Call Hierarchy (Иерархия вызовов). Это окно позволяет анализировать код на предмет того, откуда в нем вызываются методы и каким образом они связаны с другими методами. Давайте рассмотрим его возможности на примере.

Откройте клиентское приложение, созданное в предыдущем примере, и файл кода `Deck.cs`. Найдите в нем метод `Shuffle()`, щелкните правой кнопкой мыши и выберите в

контекстном меню пункт View Call Hierarchy (Просмотреть иерархию вызовов). После этого должно появиться окно, показанное на рис. 10.19 (с несколькими развернутыми разделами).

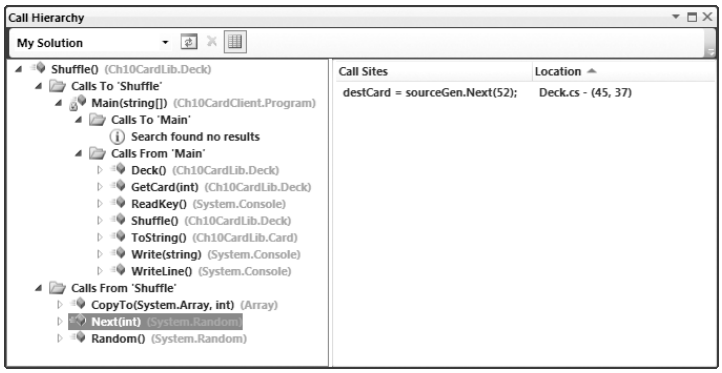


***Рис. 10.19**. Окно Call Hierarchy*

Начиная с метода `Shuffle`, в этом окне можно разворачивать узлы в дереве и просматривать код, в котором этот метод вызывается, а также все вызовы, которые производит он сам. Например, выделенный здесь метод `Next(int)` вызывается из метода `Shuffle()` и потому отображается в разделе `Calls From 'Shuffle'` (Вызовы из Shuffle). С помощью щелчка на любом вызове можно просмотреть строку кода, в которой осуществляется этот вызов. Она отображается в правой части окна вместе с информацией о месте нахождения. Двойной щелчок на информации о месте нахождения позволяет перейти к соответствующей строке кода.

Можно также разворачивать и другие методы, доступные в иерархии: на рис. 10.19 это было сделано для метода `Main()`, благодаря чему стало видно, какие вызовы производились из него.

Окно Call Hierarchy очень удобно при выполнении отладки или рефакторинга кода, поскольку позволяет оценить отношения между различными частями кода.

Резюме

В настоящей главе было завершено рассмотрение способов определения базовых классов. Разумеется, еще осталось много чего рассказать, но описанные приемы уже позволяют создавать довольно сложные приложения.

В главе было показано, как определять поля, методы и свойства, включая применение различных уровней доступа и ключевых слов в качестве модификаторов. Также рассматривались инструменты, ускоряющие получение структуры всего класса.

Кроме того, было детально описано поведение наследования и показано, как скрывать ненужные унаследованные члены с помощью ключевого слова `new` и расширять члены базового класса с использованием ключевого слова `base` вместо замены их реализации. Также рассматривались вложенные определения классов, способы определения и реализации интерфейсов, включая их явную и неявную реализацию, и разнесение определения по нескольким файлам кода с применением частичных определений классов и методов.

И, наконец, в главе был приведен пример создания и использования простой библиотеки классов, представляющих колоду игральных карт, с помощью инструмента Class Diagram (Диаграмма классов). Эта библиотека будет применяться и в последующих главах.

В следующей главе речь пойдет о другом типе классов, а именно — о коллекциях, которые также часто используются при разработке.

Упражнения

1. Напишите код, определяющий базовый класс `MyClass` с виртуальным методом `GetString`. Этот метод должен возвращать строку, хранящуюся в защищенном поле `myString`, которое доступно только через предназначенное для записи `public`-свойство `ContainedString`.
2. Создайте класс `MyDerivedClass`, унаследованный от класса `MyClass`. Переопределите в нем метод `GetString`, который возвращает строку с использованием базовой реализации данного метода, добавляя к ней еще текст `"(output from derived class)"`.
3. Частичные методы должны обязательно использовать `void` в качестве возвращаемого типа. Объясните причину.
4. Напишите код класса по имени `MyCopyableClass`, который может возвращать копию самого себя с применением метода `GetCopy`. В этом методе должен использоваться метод `MemberwiseClone`, унаследованный от `System.Object`. Добавьте в класс простое свойство и напишите клиентский код для тестирования работоспособности.
5. Напишите клиентское консольное приложение для библиотеки `Ch10CardLib`, позволяющее вытягивать сразу пять карт из перетасованной колоды (объекта `Deck`). Если все пять карт оказываются одной масти, их названия должны быть выведены на консоль вместе с текстом `Flush!`; в противном случае приложение должно завершить работу после просмотра 50 карт, с выводом текста `No flush`.

Решения упражнений приведены в приложении A.

Что вы узнали в этой главе

Тема	Основные концепции
Определение членов	В классе можно определять члены в виде полей, методов и свойств. Поле определяется с уровнем доступности, именем и типом. Метод определяется с уровнем доступности, возвращаемым типом, именем и параметрами. Свойство определяется с уровнем доступности, именем и средствами доступа `get` и/или `set`. Отдельные средства доступа к свойствам могут иметь собственные уровни доступности, которые должны быть более ограничивающими, чем уровень доступности свойства в целом.
Сокрытие и переопределение членов	Для реализации наследования свойства и методы могут определяться как абстрактные (`abstract`) или виртуальные (`virtual`) в базовом классе. В унаследованных классах абстрактные члены должны быть реализованы, а виртуальные члены могут быть переопределены с помощью ключевого слова `override`. Виртуальные члены также могут получать новые реализации с использованием ключевого слова `new`. Посредством ключевого слова `sealed` предотвращается дальнейшее переопределение виртуального члена (т.е. он запечатывается). Для обращения к базовой реализации служит ключевое слово `base`.
Реализация интерфейсов	Класс, реализующий интерфейс, должен предоставлять реализации всех общедоступных членов этого интерфейса. Интерфейсы можно реализовывать явно или неявно, при этом явные реализации доступны только через ссылку на соответствующий интерфейс.
Частичные определения	Определения классов могут быть разнесены по нескольким файлам кода с помощью ключевого слова `partial`. Посредством этого же ключевого слова можно также создавать частичные методы. Частичные методы имеют ряд ограничений, включая невозможность возврата значения и применения параметров `out`, и они не компилируются, если не предоставлена их реализация.

11

Коллекции, сравнения и преобразования

В ЭТОЙ ГЛАВЕ...

- Определение и использование коллекций
- Доступные типы коллекций
- Сравнение типов и применение операции `is`
- Сравнение значений и применение перегруженных операций
- Определение и применение преобразований
- Использование операции `as`

Все основные приемы ООП, которыми можно пользоваться в C#, уже были рассмотрены, но существует еще несколько более сложных приемов, с которыми тоже стоит ознакомиться. В настоящей главе рассматриваются следующие темы.

- **Коллекции**. Коллекции позволяют обслуживать группы объектов. В отличие от массивов, применение которых демонстрировалось в предыдущих главах, коллекции могут поддерживать более развитую функциональность, например, управлять доступом к содержащимся внутри них объектам, производить в них поиск, сортировать их и т.д. Здесь будет показано, как использовать и создавать классы коллекций, а также продемонстрированы некоторые мощные приемы для извлечения из коллекций максимальной пользы.
- **Сравнения**. При работе с объектами часто возникает необходимость сравнивать их между собой. Это особенно важно для коллекций, поскольку именно таким образом в них обеспечивается возможность выполнения сортировки. В настоящей главе будет показано, как сравнивать объекты несколькими способами, в том числе за счет перегрузки операций, а также как использовать интерфейсы `IComparable` и `IComparer` для сортировки коллекций.
- **Преобразования**. В предыдущих главах уже рассматривались преобразования объектов из одного типа в другой. В этой главе будет показано, как настраивать преобразования типов на существующие потребности.

Коллекции

В главе 5 вы ознакомились с использованием массивов для создания типов переменных, которые содержат множество объектов или значений. Однако с массивами связан ряд ограничений. Наиболее серьезным из них является то, что после создания массивы имеют фиксированный размер, из-за чего невозможно добавлять новые элементы в конец существующего массива, не создавая новый. Это часто выливается в то, что синтаксис, применяемый для манипулирования массивами, становится чрезмерно сложным. За счет применения приемов ООП можно создавать классы, которые выполняют большинство этих манипуляций внутренним образом и тем самым упрощают код, в котором используются списки элементов или массивы.

В C# массивы реализуются в виде экземпляров класса `System.Array` и представляют собой лишь один из типов классов, которые называются *классами коллекций*. В целом, классы коллекций предназначены для обслуживания списков объектов и предлагают больше функциональных возможностей, чем простые массивы. Большая часть их функциональности обеспечивается за счет реализации интерфейсов из пространства имен `System.Collections`, что делает синтаксис коллекций стандартным. В этом пространстве имен также есть и другие интересные классы, которые реализуют необходимые интерфейсы отличными от `System.Array` способами.

Поскольку функциональные возможности коллекций (включая базовые функции, такие как доступ к элементам коллекции с применением синтаксиса `[`*индекс*`]`) можно получать через интерфейсы, вы не ограничены использованием только базовых классов коллекций вроде `System.Array`. Можно также создавать собственные специализированные классы коллекций и настраивать их под потребности объектов, которые должны поставляться в виде коллекции. Одним из преимуществ такого подхода, как будет показано позже, является то, что специальные классы коллекций могут быть *строго типизированными*. Это значит, что при извлечении из коллекции элементы не понадобится приводить к правильному типу. Другое преимущество связано с возможностью предоставлять специализированные методы, которые позволяют, например, извлекать подмножество элементов. В нашем примере с колодой карт это может быть метод для получения всех элементов `Card` определенной масти.

Базовая функциональность коллекций в пространстве имен `System.Collections` предоставляется множеством интерфейсов.

- `IEnumerable` — предоставляет возможность проходить в цикле по элементам коллекции.
- `ICollection` — предоставляет возможность получать количество элементов в коллекции и копировать элементы в простой массив (унаследован от `IEnumerable`).
- `IList` — предоставляет список элементов для коллекции с возможностями доступа к этим элементам и другими базовыми операциями работы со списками (унаследован от `IEnumerable` и `ICollection`).
- `IDictionary` — подобен `IList`, но предоставляет список элементов, доступных по значению ключа, а не по индексу (унаследован от `IEnumerable` и `ICollection`).

Класс `System.Array` реализует интерфейсы `IList`, `ICollection` и `IEnumerable`. Однако он не поддерживает некоторую более совершенную функциональность `IList` и представляет список элементов с использованием фиксированного размера.

Использование коллекций

Один из классов в пространстве имен `Systems.Collections` — `System.Collections.ArrayList` — тоже реализует интерфейсы `IList`, `ICollection` и `IEnumerable`, но делает это более совершенным образом, чем класс `System.Array`. В то время как массивы фиксированы по размеру (ни добавлять, ни удалять из них элементы не допускается), этот класс позволяет представлять список элементов переменной длины. В следующем практическом занятии демонстрируется пример применения этого класса и простого массива.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Массивы или более совершенные коллекции

1. Создайте новый проект типа консольного приложения по имени `Ch11Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter11`.
2. Добавьте в него три новых класса с именами `Animal` (Животное), `Cow` (Корова) и `Chicken` (Курица), щелкнув в окне Solution Explorer правой кнопкой мыши и выбрав в контекстном меню пункт Add⇨Class (Добавить⇨Класс) для каждого класса.
3. Измените код в файле `Animal.cs`, как показано ниже:

```
namespace Ch11Ex01
{
  public abstract class Animal
  {
    protected string name;
    public string Name
    {
      get
      {
        return name;
      }
      set
      {
        name = value;
      }
    }
```

```
    public Animal()
    {
      name = "The animal with no name";
           // Животное без клички
    }
    public Animal(string newName)
    {
      name = newName;
    }
    public void Feed()
    {
       Console.WriteLine("{0} has been fed.", name);
                    // Вывод информации о кормлении
    }
  }
}
```

Фрагмент кода ***Ch11Ex01\Animal.cs***

4. Измените код в файле `Cow.cs` следующим образом:

```
namespace Ch11Ex01
{
  public class Cow : Animal
  {
    public void Milk()
    {
      Console.WriteLine("{0} has been milked.", name);
                    // Вывод информации о доении
    }
    public Cow(string newName) : base(newName)
    {
    }
  }
}
```

Фрагмент кода ***Ch11Ex01\Cow.cs***

5. Измените код в файле `Chicken.cs`, как показано ниже:

```
namespace Ch11Ex01
{
  public class Chicken : Animal
  {
    public void LayEgg()
    {
      Console.WriteLine("{0} has laid an egg.", name);
                    // Вывод информации о несении яиц
    }

    public Chicken(string newName) : base(newName)
    {
    }
  }
}
```

Фрагмент кода ***Ch11Ex01\Chicken.cs***

6. Измените код в файле `Program.cs` следующим образом:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
```

```
  using System.Text;
namespace Ch11Ex01
{
 class Program
 {
  static void Main(string[] args)
  {
   Console.WriteLine("Create an Array type collection of Animal " +
                     "objects and use it:");
                    // Создание коллекции типа Array из объектов
                    // Animal и ее использование
    Animal[] animalArray = new Animal[2];
    Cow myCow1 = new Cow("Deirdre");
    animalArray[0] = myCow1;
    animalArray[1] = new Chicken("Ken");

    foreach (Animal myAnimal in animalArray)
    {
      Console.WriteLine("New {0} object added to Array collection, " +
                        "Name = {1}", myAnimal.ToString(), myAnimal.Name);
                    // В коллекцию Array добавлен новый объект
    }
    Console.WriteLine("Array collection contains {0} objects.",
                    // Вывод количества объектов в коллекции Array

    animalArray.Length);
    animalArray[0].Feed();
    ((Chicken)animalArray[1]).LayEgg();
    Console.WriteLine();

    Console.WriteLine("Create an ArrayList type collection of Animal " +
                      "objects and use it:");
                    // Создание коллекции объектов Animal типа ArrayList
                    // и ее использование
    ArrayList animalArrayList = new ArrayList();
    Cow myCow2 = new Cow("Hayley");
    animalArrayList.Add(myCow2);
    animalArrayList.Add(new Chicken("Roy"));
    foreach (Animal myAnimal in animalArrayList)
    {
      Console.WriteLine("New {0} object added to ArrayList collection," +
                        " Name = {1}", myAnimal.ToString(), myAnimal.Name);
                    // Новый объект {0} был добавлен в коллекцию ArrayList
    }
    Console.WriteLine("ArrayList collection contains {0} objects.",
                      animalArrayList.Count);
                    // В коллекции ArrayList содержится {0} объектов
    ((Animal)animalArrayList[0]).Feed();
    ((Chicken)animalArrayList[1]).LayEgg();
    Console.WriteLine();
    Console.WriteLine("Additional manipulation of ArrayList:");
                    // Дополнительное манипулирование коллекцией ArrayList
    animalArrayList.RemoveAt(0);
    ((Animal)animalArrayList[0]).Feed();
    animalArrayList.AddRange(animalArray);
    ((Chicken)animalArrayList[2]).LayEgg();
    Console.WriteLine("The animal called {0} is at index {1}.",
                      myCow1.Name, animalArrayList.IndexOf(myCow1));
                    // Вывод животных и позиций
```

```
            myCow1.Name = "Janice";
            Console.WriteLine("The animal is now called {0}.",
                        ((Animal)animalArrayList[1]).Name);
                      // Вывод животных
            Console.ReadKey();
        }
    }
}
```

Фрагмент кода ***Ch11Ex01\Program.cs***

7. Запустите приложение. На рис. 11.1 показан результат, который должен получиться.

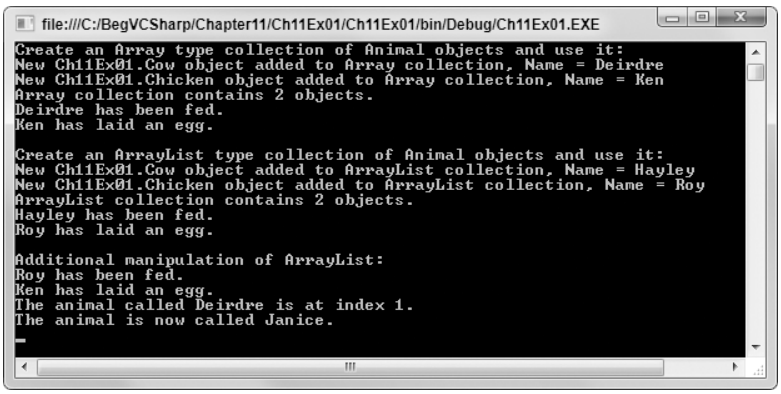


Рис. 11.1. *Приложение* `Ch11Ex01` *в действии*

Описание работы

В этом примере сначала создаются две коллекции объектов: в первой используется класс `System.Array` (простой массив), а во второй — `System.Collections.ArrayList`. Обе коллекции состоят из объектов `Animal`, которые определяются в файле `Animal.cs`. Класс `Animal` является абстрактным, так что создавать его экземпляры нельзя, хотя иметь в коллекции элементы, являющиеся экземплярами классов `Cow` и `Chicken`, которые наследуются от класса `Animal`, можно за счет применения полиморфизма, о котором рассказывалось в главе 8.

После создания массивов внутри метода `Main()` в `Class1.cs` с ними производятся различные манипуляции для демонстрации их характеристик и возможностей. Некоторые из выполняемых операций подходят как для коллекции `Array`, так и для коллекции `ArrayList`, хотя их синтаксис немного отличается. Однако есть операции, выполнение которых возможно только для более совершенного типа `ArrayList`.

Сначала давайте рассмотрим похожие операции и сравним код и результаты обоих типов коллекций. В первую очередь возьмем операцию создания коллекции. Простой массив перед использованием должен быть инициализирован с фиксированным размером. В случае массива `animalArray` это делается с применением стандартного синтаксиса, который был показан в главе 5:

```
Animal[] animalArray = new Animal[2];
```

Для коллекций `ArrayList` указывать их размер при инициализации не требуется, что позволяет создать список `animalArrayList` следующим образом:

```
ArrayList animalArrayList = new ArrayList();
```

С этим классом можно использовать и два других конструктора. Первый позволяет копировать содержимое существующей коллекции в новый экземпляр за счет указания существующей коллекции в качестве параметра, а второй — задавать *емкость* коллекции, тоже путем указания соответствующего значения в качестве параметра. Емкость, указанная в виде значения `int`, отвечает за то, сколько изначально элементов может содержаться в коллекции. Абсолютной она, однако, не является, поскольку будет автоматически удваиваться, если количество элементов превысит ее значение.

В случае массивов ссылочных типов (таких как `Animal` и производные от него) простая инициализация с указанием размера не приводит к инициализации содержащихся внутри элементов. Перед использованием элемент должен быть инициализирован, а это значит, что элементам должны присваиваться инициализированные объекты:

```
Cow myCow1 = new Cow("Deirdre");
animalArray[0] = myCow1;
animalArray[1] = new Chicken("Ken");
```

В показанном выше коде это делается двумя способами: один раз за счет присваивания существующего объекта `Cow`, а второй раз — путем создания нового объекта `Chicken`. Главное отличие здесь в том, что первый метод предусматривает создание ссылки на объект в массиве, которая используется чуть позже в этом же коде.

В случае коллекции `ArrayList` никаких существующих элементов не доступно и даже таких, которые ссылаются на `null`. Это значит, что присваивать новые экземпляры элементам таким же способом нельзя. Для добавления новых элементов используется метод `Add()` объекта `ArrayList`:

```
Cow myCow2 = new Cow("Hayley");
animalArrayList.Add(myCow2);
animalArrayList.Add(new Chicken("Roy"));
```

Не считая немного отличающегося синтаксиса, подобным образом в коллекцию можно добавлять как новые, так и существующие объекты. После добавления элементы можно перезаписывать с применением синтаксиса, идентичного массивам:

```
animalArrayList[0] = new Cow("Alma");
```

В данном примере это, однако, не делается.

В главе 5 было показано, как использовать структуру `foreach` для прохода по массиву. Это возможно потому, что в классе `System.Array` реализован интерфейс `IEnumerable`, а единственный доступный в этом интерфейсе метод `GetEnumerator()` позволяет проходить циклом по элементам коллекции. Более подробно об этом речь пойдет позже в настоящей главе. Что касается рассматриваемого примера, то здесь структура `foreach` применяется для вывода информации обо всех объектах `Animal` в массиве:

```
foreach (Animal myAnimal in animalArray)
{
  Console.WriteLine("New {0} object added to Array collection, " +
                    "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

Объект `ArrayList` также поддерживает интерфейс `IEnumerable` и может использоваться со структурой `foreach`. В данном случае синтаксис выглядит идентично:

```
foreach (Animal myAnimal in animalArrayList)
{
  Console.WriteLine("New {0} object added to ArrayList collection, " +
                    "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

Далее для вывода информации о количестве элементов в массиве используется свойство `Length`:

```
Console.WriteLine("Array collection contains {0} objects.",
                  animalArray.Length);
```

В случае коллекции ArrayList можно добиться аналогичного поведения, но только за счет применения свойства Count, которое является частью интерфейса ICollection:

```
Console.WriteLine("ArrayList collection contains {0} objects.",
                  animalArrayList.Count);
```

Коллекции, будь то простые массивы или более сложные коллекции, не особо полезны, если не предоставляют доступа к содержащимся внутри них элементам. Простые массивы являются строго типизированными, т.е. позволяют напрямую получать доступ к типу элементов, которые содержатся внутри них. Это означает, что методы их элементов можно вызывать напрямую:

```
animalArray[0].Feed();
```

В данном случае типом массива является абстрактный тип Animal, поэтому вызывать напрямую методы, предоставляемые производными от него классами, нельзя. Вместо этого нужно использовать синтаксис приведения типов:

```
((Chicken)animalArray[1]).LayEgg();
```

ArrayList — это коллекция объектов System.Object (объекты Animal были присвоены ей посредством полиморфизма). Это означает, что синтаксис приведения должен использоваться для всех элементов:

```
((Animal)animalArrayList[0]).Feed();
((Chicken)animalArrayList[1]).LayEgg();
```

В остальной части кода иллюстрируются некоторые возможности коллекции ArrayList, которые отсутствуют у Array. Первым делом, ArrayList позволяет удалять элементы с помощью методов Remove() и RemoveAt(), которые являются частью реализации интерфейса IList. Элементы удаляются из массива на основе, соответственно, ссылки на удаляемый элемент либо его индекса. В данном примере используется второй метод для удаления первого элемента в списке, которым будет объект Cow, имеющий в свойстве Name значение Hayley:

```
animalArrayList.RemoveAt(0);
```

Поскольку локальная ссылка на объект уже имеется, в качестве альтернативы можно было бы применить такую строку кода:

```
animalArrayList.Remove(myCow2);
```

И обоих случаях единственным элементом, который остается в коллекции, будет объект Chicken, доступ к которому получается следующим образом:

```
((Animal)animalArrayList[0]).Feed();
```

Любые изменения, вносимые в элементы ArrayList, в результате которых в массиве остается N элементов, осуществляются так, чтобы индексы были в диапазоне от 0 до N-1. Например, удаление элемента с индексом 0 приводит к сдвигу всех остальных элементов в массиве на одну позицию, из-за чего доступ к объекту Chicken будет осуществляться с использованием индекса 0, а не 1. Элемента с индексом 1 больше не существует (т.к. изначально элементов было всего два), а это значит, что приведенная ниже строка кода вызовет исключение:

```
((Animal)animalArrayList[1]).Feed();
```

Коллекции ArrayList позволяют добавлять сразу несколько элементов с помощью метода AddRange(). Этот метод может принимать любой объект с интерфейсом ICollection, включая созданный ранее массив animalArray:

```
animalArrayList.AddRange(animalArray);
```

Чтобы удостовериться в его работоспособности, можно попробовать получить доступ к третьему элементу в коллекции, который будет вторым элементом в animalArray:

```
((Chicken)animalArrayList[2]).LayEgg();
```

Метод AddRange() не является частью предоставляемых ArrayList интерфейсов. Он принадлежит классу ArrayList и демонстрирует тот факт, что в своих классах коллекций можно обеспечивать специализированное поведение, выходящее за рамки того, которого требуют рассмотренные ранее интерфейсы. В этом классе предлагаются и другие интересные методы, такие как InsertRange(), позволяющий вставлять массив объектов в любое место внутри списка, и методы, которые решают задачи, подобные сортировке и переупорядочиванию массива.

И, наконец, в примере показано, что можно иметь несколько ссылок на один и тот же объект. Метод IndexOf() (часть интерфейса IList) позволяет увидеть не только то, что myCow1 (объект, который изначально добавлялся в animalArray) теперь является частью коллекции animalArrayList, но и его индекс:

```
Console.WriteLine("The animal called {0} is at index {1}.",
                  myCow1.Name, animalArrayList.IndexOf(myCow1));
```

В следующих двух строках демонстрируется переименование объекта через ссылку на объект и вывод нового имени с использованием ссылки на коллекцию:

```
myCow1.Name = "Janice";
Console.WriteLine("The animal is now called {0}.",
             ((Animal)animalArrayList[1]).Name);
```

Определение коллекций

Теперь, когда известны возможности, предлагаемые более совершенными классами коллекций, пришла пора научиться создавать собственные строго типизированные коллекции. Одним из возможных способов является реализация всех требуемых методов вручную, но это может оказаться очень длинным и сложным процессом. Существует, однако, и альтернативный вариант — наследование коллекции от класса вроде System.Collections.CollectionBase, который является абстрактным и предоставляет большую часть реализации коллекции автоматически. Настоятельно рекомендуется использовать именно этот вариант.

В классе CollectionBase поддерживаются такие интерфейсы, как IEnumerable, ICollection и IList, но предоставляется лишь часть требуемой для них реализации: методы Clear() и RemoveAt() для интерфейса IList и свойство Count для интерфейса ICollection. Все остальные детали должны быть реализованы самостоятельно.

Для упрощения этого процесса в CollectionBase предлагаются два защищенных свойства, которые предоставляют доступ к хранимым объектам: List для доступа к элементам через интерфейс IList и InnerList для доступа к объекту ArrayList, используемому для хранения элементов.

Например, базовые детали класса коллекции для хранения объектов Animal можно было бы определить следующим образом (более полная реализация будет показана позже):

```
public class Animals : CollectionBase
{
  public void Add(Animal newAnimal)
  {
    List.Add(newAnimal);
  }
  public void Remove(Animal oldAnimal)
  {
    List.Remove(oldAnimal);
  }
```

```
  public Animals()
  {
  }
}
```

Здесь `Add()` и `Remove()` были реализованы как строго типизированные методы, предусматривающие применение для получения доступа к элементам стандартного метода `Add()` из интерфейса `IList`. Теперь они будут работать только с классами `Animal` или классами, унаследованными от `Animal`, что отличается от приводимых ранее реализаций `ArrayList`, где они могли работать с любым объектом.

Класс `CollectionBase` позволяет использовать синтаксис `foreach` с производными коллекциями, например:

```
Console.WriteLine("Using custom collection class Animals:");
                // Использование специальной коллекции Animals
Animals animalCollection = new Animals();
animalCollection.Add(new Cow("Sarah"));
foreach (Animal myAnimal in animalCollection)
{
  Console.WriteLine("New {0} object added to custom collection, " +
                "Name = {1}", myAnimal.ToString(), myAnimal.Name);
               //Новый объект {0} был добавлен в специальную коллекцию
}
```

Однако применять следующий синтаксис не допускается:

```
animalCollection[0].Feed();
```

Для получения доступа к элементам по их индексам подобным образом должен использоваться индексатор.

Индексаторы

Индексатор (indexer) представляет собой свойство особого вида, которое можно добавлять в класс для обеспечения доступа к элементам как в массивах. На самом деле с помощью индексатора можно обеспечить и гораздо более сложный доступ, поскольку вместе с ним в квадратных скобках допускается также определять и использовать параметры любых сложных типов. Тем не менее, наиболее распространенным способом его применения является реализация простого числового индекса для элементов.

Ниже показано, как добавить индексатор в коллекцию `Animals`, состоящую из объектов `Animal`:

```
public class Animals : CollectionBase
{
  ...
  public Animal this[int animalIndex]
  {
    get
    {
      return (Animal)List[animalIndex];
    }
    set
    {
       List[animalIndex] = value;
    }
  }
}
```

Здесь используется ключевое слово `this` вместе с параметром в квадратных скобках, но весь остальной код выглядит во многом так же, как и код любого другого свойства. Такой

синтаксис вполне логичен, поскольку позволяет получать доступ к индексатору с использованием имени объекта со следующим ним в квадратных скобках индексным параметром или параметрами (например, `MyAnimals[0]`).

В следующем коде индексатор применяется в отношении свойства `List` (т.е. в отношении интерфейса `IList`, который предоставляет в `CollectionBase` доступ к коллекции `ArrayList`, хранящей элементы):

```
return (Animal)List[animalIndex];
```

Явное приведение является здесь *обязательным*, поскольку свойство `IList.List` возвращает объект `System.Object`. Обратите внимание, что для данного индексатора определен тип. Именно этот тип и будет получен при доступе к элементу с использованием данного индексатора. Благодаря такой строгой типизации, можно записать следующую строку кода:

```
animalCollection[0].Feed();
```

вместо такой:

```
((Animal)animalCollection[0]).Feed();
```

Это демонстрирует еще одну полезную возможность строго типизированных специальных коллекций. В следующем практическом занятии будет расширен предыдущий пример для демонстрации всех описанных приемов на практике.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Реализация коллекции `Animals`

1. Создайте новое консольное приложение по имени `Ch11Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter11`.
2. Щелкните правой кнопкой мыши на имени этого проекта в окне Solution Explorer и выберите в контекстном меню пункт Add⇨Add Existing Item (Добавить⇨Существующий элемент).
3. Выберите из каталога `C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01` файлы `Animal.cs`, `Cow.cs` и `Chicken.cs` и щелкните на кнопке Add (Добавить).
4. Измените объявление пространства имен в каждом из трех добавленных файлов следующим образом:

```
namespace Ch11Ex02
```

*Фрагмент кода **Ch11Ex02\Animal.cs**, **Ch11Ex02\Cow.cs** и **Ch11Ex02\Chicken.cs***

5. Добавьте новый класс по имени `Animals`.
6. Измените код в файле `Animals.cs`, как показано ниже:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex02
{
  public class Animals : CollectionBase
  {
    public void Add(Animal newAnimal)
    {
      List.Add(newAnimal);
    }
```

```
    public void Remove(Animal newAnimal)
    {
      List.Remove(newAnimal);
    }
    public Animals()
    {
    }
    public Animal this[int animalIndex]
    {
      get
      {
        return (Animal)List[animalIndex];
      }
      set
      {
        List[animalIndex] = value;
      }
    }
  }
}
```

*Фрагмент кода **Ch11Ex02\Animals.cs***

7. Измените код в файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  Animals animalCollection = new Animals();
  animalCollection.Add(new Cow("Jack"));
  animalCollection.Add(new Chicken("Vera"));
  foreach (Animal myAnimal in animalCollection)
  {
    myAnimal.Feed();
  }
  Console.ReadKey();
}
```

*Фрагмент кода **Ch11Ex02\Program.cs***

8. Запустите приложение. На рис. 11.2 показан результат, который должен получиться.

***Рис. 11.2.** Приложение `Ch11Ex02` в действии*

Описание работы

В этом примере на основе кода из предыдущего раздела реализована строго типизированная коллекция объектов `Animal` в классе по имени `Animals`. В коде метода `Main()` просто создается экземпляр объекта `Animals` по имени `animalCollection`, в него добавляются два элемента (экземпляры `Cow` и `Chicken`) и в цикле `foreach` вызывается метод `Feed()`, который оба объекта наследуют от своего базового класса `Animal`.

Добавление коллекции `Cards` в `CardLib`

В предыдущей главе мы создали проект библиотеки классов `Ch10CardLib`, в который включили класс `Card`, представляющий игральную карту, и класс `Deck`, представляющий колоду карт, т.е. коллекцию классов `Card`. Эта коллекция была реализована в виде простого массива.

В настоящей главе мы переименуем эту библиотеку в Ch11CardLib и добавим в нее новый класс. Этот новый класс — Cards — будет специальной коллекцией объектов Card, которая обеспечит все преимущества, описанные ранее в этой главе. Создайте новый проект библиотеки классов по имени Ch11CardLib в каталоге C:\BegVCSharp\Chapter11, удалите из него автоматически сгенерированный файл Class1.cs, выберите в меню Project (Проект) пункт Add Existing Item (Добавить существующий элемент), найдите в каталоге C:\BegVCSharp\Chapter10\Ch10CardLib\Ch10CardLib файлы Card.cs, Deck.cs, Suit.cs и Rank.cs и добавьте их в новый проект. Как и в прошлый раз, чрезмерно детальные шаги здесь приводиться не будут. При желании можно воспользоваться готовой версией проекта, которая входит в состав кода примеров для этой главы.

При копировании исходных файлов из Ch10CardLib в Ch11CardLib не забудьте изменить объявления пространств имен, чтобы они ссылались на Ch11CardLib. То же самое касается и консольного приложения Ch10CardClient, которое будет использоваться для тестирования.

В состав кода примеров для этой главы входит проект, содержащий весь код, который может потребоваться для разнообразных расширений Ch11CardLib. Этот код поделен на разделы, и при желании испробовать их в действии достаточно просто удалить из них комментарии.

Чтобы построить данный проект самостоятельно, добавьте в него новый класс Cards и модифицируйте код в представляющем его файле Cards.cs, как показано ниже:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11CardLib
{
  public class Cards : CollectionBase
  {
    public void Add(Card newCard)
    {
      List.Add(newCard);
    }
    public void Remove(Card oldCard)
    {
      List.Remove(oldCard);
    }
    public Cards()
    {
    }

    public Card this[int cardIndex]
    {
      get
      {
        return (Card)List[cardIndex];
      }
      set
      {
        List[cardIndex] = value;
      }
    }
```

```
    /// <summary>
    /// Вспомогательный метод для копирования экземпляров карт в другой
    /// экземпляр Cards — используемый в Deck.Shuffle(). В данной реализации
    /// предполагается, что размеры исходной и целевой коллекций совпадают.
    /// </summary>
    public void CopyTo(Cards targetCards)
    {
      for (int index = 0; index < this.Count; index++)
      {
        targetCards[index] = this[index];
      }
    }

    /// <summary>
    /// Выполнение проверки на предмет наличия в коллекции определенной карты.
    /// Эта проверка предусматривает вызов метода Contains из
    /// ArrayList для коллекции, к которой получается
    /// доступ через свойство InnerList.
    /// </summary>
    public bool Contains(Card card)
    {
      return InnerList.Contains(card);
    }
  }
}
```

Фрагмент кода ***Ch11CardLib\Cards.cs***

Затем измените код в файле `Deck.cs` так, чтобы в нем использовалась новая коллекция, а не массив:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11CardLib
{
  public class Deck
  {
    private Cards cards = new Cards();
    public Deck()
    {
      // Здесь была удалена одна строка кода
      for (int suitVal = 0; suitVal < 4; suitVal++)
      {
        for (int rankVal = 1; rankVal < 14; rankVal++)
        {
          cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
        }
      }
    }

    public Card GetCard(int cardNum)
    {
      if (cardNum >= 0 && cardNum <= 51)
        return cards[cardNum];
      else
        throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
            "Value must be between 0 and 51."));
            //Значение может находиться только в диапазоне от 0 до 51
      }
```

```
    public void Shuffle()
    {
      Cards newDeck = new Cards();
      bool[] assigned = new bool[52];
      Random sourceGen = new Random();
      for (int i = 0; i < 52; i++)
      {
        int sourceCard = 0;
        bool foundCard = false;
        while (foundCard == false)
      {
        sourceCard = sourceGen.Next(52);
        if (assigned[sourceCard] == false)
            foundCard = true;
      }
        assigned[sourceCard] = true;
        newDeck.Add(cards[sourceCard]);
      }
      newDeck.CopyTo(cards);
    }
  }
}
```

Фрагмент кода ***Ch11CardLib\Deck.cs***

Как видите, изменений не очень много. Большинство из них связано с модификацией логики тасования: теперь необходимо добавлять карту в начало новой коллекции `Cards` по имени `newDeck` из позиции со случайным индексом в `cards`.

Теперь можно использовать клиентское консольное приложение `Ch10CardClient`, которое создавалось для решения `Ch10CardLib`, вместе с новой библиотекой. Результат будет выглядеть точно так же, поскольку сигнатуры методов `Deck` не изменились. Однако теперь клиенты библиотеки могут работать с классом коллекции `Cards`, а не полагаться на массивы объектов `Card` в случае, например, определения имеющихся на руках карт в приложении для карточной игры.

Коллекции с ключами и интерфейс `IDictionary`

Вместо интерфейса `IList` для коллекций можно также реализовать похожий интерфейс `IDictionary`, который позволяет получать доступ к элементу по значению ключа (такому как строковое имя), а не по числовому индексу. Это также достигается с использованием индексатора, хотя в таком случае в качестве параметра индексатора должен быть указан ключ, который ассоциируется с хранимым элементом, а не индекс типа `int`. В результате коллекция становится гораздо более дружественной к пользователю.

Как и в случае индексированных коллекций, имеется базовый класс, упрощающий реализацию интерфейса `IDictionary` — `DictionaryBase`. Этот класс также реализует интерфейсы `IEnumerable` и `ICollection`, предоставляя базовые возможности для манипулирования, общие для любой коллекции.

В классе `DictionaryBase`, как и в `CollectionBase`, реализованы некоторые (но не все) члены, полученные от поддерживаемых им интерфейсов. Подобно `CollectionBase`, реализованы члены `Clear` и `Count`, но не `RemoveAt()`, поскольку этот член является методом из интерфейса `IList` и в интерфейсе `IDictionary` отсутствует. Однако в интерфейсе `IDictionary` есть метод `Remove()`, который должен быть реализован в классе специальной коллекции на основе `DictionaryBase`.

В следующем коде показана альтернативная версия класса `Animals`, на этот раз унаследованная от `DictionaryBase`. Она включает реализации методов `Add()` и `Remove()`, а также индексатора с доступом по ключу.

```
public class Animals : DictionaryBase
{
  public void Add(string newID, Animal newAnimal)
  {
    Dictionary.Add(newID, newAnimal);
  }
  public void Remove(string animalID)
  {
    Dictionary.Remove(animalID);
  }
  public Animals()
  {
  }
  public Animal this[string animalID]
  {
    get
    {
       return (Animal)Dictionary[animalID];
    }
    set
    {
      Dictionary[animalID] = value;
    }
  }
}
```

Ниже описаны отличия между этими членами.

- **`Add()`**. Принимает два параметра — ключ и значение — для совместного хранения. Словарная коллекция имеет член по имени `Dictionary`, который наследуется от `DictionaryBase` и представляет собой интерфейс `IDictionary`. Этот интерфейс имеет собственный метод `Add()`, который принимает в качестве параметров два объекта. В показанной реализации он принимает строковое значение в качестве ключа и объект `Animal` в качестве данных, подлежащих сохранению вместе с этим ключом.
- **`Remove()`**. Принимает в качестве параметра ключ, а не ссылку на объект, и удаляет элемент с указанным значением ключа.
- **Индексатор**. Использует строковое значение ключа, а не индекс, и получает с его помощью доступ к хранимому элементу через унаследованный член `Dictionary`. Приведение типов здесь является обязательным.

Еще одним отличием между коллекциями на основе `DictionaryBase` и коллекциями на базе `CollectionBase` является то, что структура `foreach` в них работает несколько поразному. В классе коллекции из предыдущего раздела внутри `foreach` можно было извлекать объекты `Animal` прямо из коллекции. Здесь же применение `foreach` с производным классом `DictionaryBase` приводит к получению структур `DictionaryEntry` — еще одного типа из пространства имен `System.Collections`. Из-за этого для получения самих объектов `Animal` должен использоваться член `Value` этой структуры, а для ассоциированных с ними ключей — член `Key`. Прежний код:

```
foreach (Animal myAnimal in animalCollection)
{
  Console.WriteLine("New {0} object added to custom collection, " +
                    "Name = {1}", myAnimal.ToString(), myAnimal.Name);
                   // Новый объект {0} был добавлен в специальную коллекцию
}
```

должен быть заменен следующим эквивалентным кодом:

```
foreach (DictionaryEntry myEntry in animalCollection)
{
  Console.WriteLine("New {0} object added to custom collection, " +
                    "Name = {1}", myEntry.Value.ToString(),
                    ((Animal)myEntry.Value).Name);
                   // Новый объект {0} был добавлен в специальную коллекцию
}
```

Это поведение можно переопределить так, чтобы доступ к объектам `Animal` получался напрямую через `foreach`. Для этого существует несколько способов, самым простым из которых является реализация итератора.

Итераторы

Ранее в этой главе было показано, что интерфейс `IEnumerable` предоставляет возможность использовать циклы `foreach`. В циклах `foreach` часто выгодно работать со своими классами, а не только с классами коллекций, которые были описаны в предыдущих разделах.

Тем не менее, переопределение этого поведения, или предоставление собственной реализации для него, не всегда оказывается простым. Давайте сначала более внимательно посмотрим, как работает цикл `foreach`. Ниже описано, что на самом деле происходит в цикле `foreach`, осуществляющем проход по коллекции `collectionObject`.

1. Вызывается метод `collectionObject.GetEnumerator()`, который возвращает ссылку на `IEnumerator`. Этот метод доступен через реализацию интерфейса `IEnumerable`, хотя она является необязательной.
2. На возвращенном интерфейсе вызывается метод `MoveNext()`.
3. Если метод `MoveNext()` возвращает `true`, с помощью свойства `Current` интерфейса `IEnumerator` получается ссылка на объект, которая используется в цикле `foreach`.
4. Два последних шага повторяются до тех пор, пока `MoveNext()` не вернет `false`, после чего цикл завершается.

Для обеспечения такого поведения в своих классах вы должны переопределить несколько методов, отслеживать индексы, поддерживать свойство `Current` и т.д., т.е. приложить немало усилий для достижения относительно небольшого эффекта.

Более простой альтернативой является использование итератора. Применение итераторов, по сути, приводит к автоматической генерации большого объема кода "за кулисами" с надлежащей привязкой к нему. Синтаксис использования операторов также гораздо более прост в освоении.

Удачным определением итератора может служить следующее: это блок кода, который предоставляет все значения, подлежащие использованию в блоке `foreach`, по очереди. Обычно в роли этого блока кода выступает метод, хотя в качестве итератора также может применяться блок доступа к свойству или какой-то другой блок кода. Для простоты здесь будут рассматриваться только методы.

Каким бы не был этот блок кода, количество возможных кандидатов на роль его возвращаемого типа является ограниченным. Вопреки возможным ожиданиям, этот возвращаемый тип не может совпадать с типом подвергаемого перечислению объекта. Например, в классе, представляющем коллекцию объектов `Animal`, типом, возвращаемым из блока итератора, `Animal` быть не может. Двумя допустимыми кандидатами на роль возвращаемого значения являются типы упоминавшихся ранее интерфейсов `IEnumerable` или `IEnumerator`. Использовать их нужно следующим образом.

- Для итерации по классу следует применять метод `GetEnumerator()` с возвращаемым типом `IEnumerator`.
- Для итерации по члену класса, например, методу, должен применяться `IEnumerable`.

Внутри блока итератора выбор используемых в цикле `foreach` значений должен производиться с помощью ключевого слова `yield`. Ниже показан необходимый синтаксис:

```
yield return <значение>;
```

Этого вполне достаточно для построения очень простого примера:

```
public static IEnumerable SimpleList()
{
  yield return "string 1";
  yield return "string 2";
  yield return "string 3";
}
public static void Main(string[] args)
{
  foreach (string item in SimpleList())
    Console.WriteLine(item);
  Console.ReadKey();
}
```

Фрагмент кода ***SimpleIterators\Program.cs***

При тестировании данного кода не забудьте добавить оператор `using` для пространства имен `System.Collections` или же полностью указывайте имя интерфейса — `System.Collections.IEnumerable`. В качестве альтернативы можно воспользоваться кодом из проекта `SimpleIterators`, входящего в состав примеров для этой главы.

Здесь блоком итератора является статический метод `SimpleList()`. Поскольку это метод, возвращаемым типом служит `IEnumerable`. В `SimpleList()` используется ключевое слово `yield` для предоставления блоку `foreach` трех значений, каждое из которых затем выводится на консоль. На рис. 11.3 показан результат выполнения данного кода.

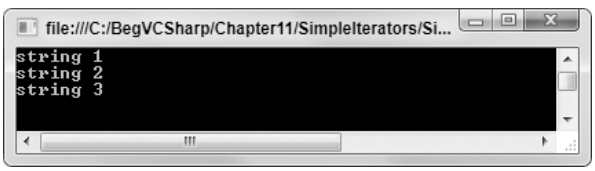


Рис. 11.3. *Метод-итератор `SimpleList()` в действии*

Очевидно, что это не особо полезный итератор, однако он демонстрирует, насколько просто может выглядеть реализация. При просмотре кода может возникнуть вопрос: откуда он знает, что нужно возвращать элементы типа `string`? На самом деле код этого не знает и возвращает объекты типа `object`, а `object`, как известно, является базовым классом для всех типов, благодаря чему из операторов `yield` можно возвращать все что угодно.

Тем не менее, компилятор достаточно интеллектуален, чтобы интерпретировать возвращаемые значения как относящиеся к желаемому типу в контексте цикла `foreach`. Здесь в коде запрашиваются значения типа `string`, поэтому именно они и предоставляются для работы. Если изменить одну из строк `yield` так, чтобы она возвращала, скажем, целое число, сгенерируется исключение, указывающее на неправильное приведение типов в цикле `foreach`.

Об итераторах следует знать еще один момент. В них можно прерывать возврат информации циклу `foreach` с помощью следующего оператора:

```
yield break;
```

Этот оператор прекращает обработку итератора, а также выполнение использующего его цикла `foreach`.

Давайте рассмотрим более сложный пример. В следующем практическом занятии будет реализован итератор, который получает простые числа.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Реализация итератора

1. Создайте новое консольное приложение по имени `Ch11Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter11`.
2. Добавьте новый класс `Primes` и измените его код следующим образом:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Ch11Ex03
{
  public class Primes
  {
    private long min;
    private long max;
    public Primes() : this(2, 100)
    {
    }
    public Primes(long minimum, long maximum)
    {
      if (min < 2)
        min = 2;
      else
        min = minimum;
      max = maximum;
    }
    public IEnumerator GetEnumerator()
    {
      for (long possiblePrime = min; possiblePrime <= max; possiblePrime++)
      {
        bool isPrime = true;
        for (long possibleFactor = 2; possibleFactor <=
         (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)
        {
          long remainderAfterDivision = possiblePrime % possibleFactor;
          if (remainderAfterDivision == 0)
          {
            isPrime = false;
            break;
          }
        }
        if (isPrime)
        {
          yield return possiblePrime;
        }
      }
    }
  }
}
```

Фрагмент кода ***Ch11Ex03\Primes.cs***

3. Измените код в файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  Primes primesFrom2To1000 = new Primes(2, 1000);
  foreach (long i in primesFrom2To1000)
    Console.Write("{0} ", i);
  Console.ReadKey();
}
```

Фрагмент кода ***Ch11Ex03\Program.cs***

4. Запустите приложение. На рис. 11.4 показан результат, который должен получиться.

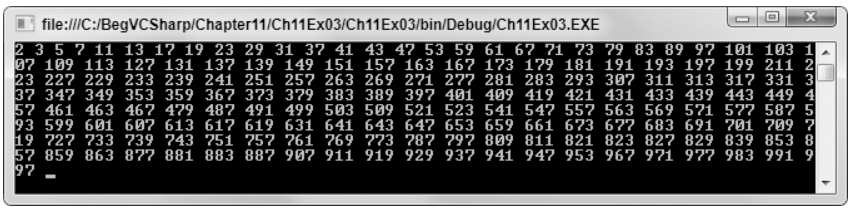


Рис. 11.4. *Приложение `Ch11Ex03` в действии*

Описание работы

В примере построен класс `Primes`, который позволяет проходить по коллекции простых чисел, находящихся между определенным верхним и нижним пределом. Этот класс инкапсулирует простые числа и для предоставления своей функциональности использует итератор.

Код класса `Primes` начинается с определения базовых деталей: двух полей для хранения максимального и минимального значений, между которыми должен выполняться поиск, и конструкторов для установки этих значений. Обратите внимание, что на минимальное значение накладывается ограничение, не позволяющее, чтобы оно было меньше 2. В таком ограничении действительно есть смысл, т.к. 2 является наименьшим простым числом. Самый интересный код находится в методе `GetEnumerator()`. Сигнатура этого метода отвечает правилам оформления блока итератора, поскольку предусматривает возврат типа `IEnumerator`:

```
public IEnumerator GetEnumerator()
{
```

Для извлечения простых чисел в определенных пределах необходимо проверять каждое число по очереди, поэтому сначала идет цикл `for`:

```
for (long possiblePrime = min; possiblePrime <= max; possiblePrime++)
{
```

Вначале предполагается, что число является простым, а затем выполняется проверка, действительно ли это так. Эта проверка подразумевает выяснение того, существует ли в промежутке между 2 и значением квадратного корня проверяемого числа такое значение, на которое проверяемое число делится без остатка. Если да, значит, это число не является простым и потому происходит переход к следующему числу. Если же число действительно оказывается простым, тогда оно передается в цикл `foreach` с применением оператора `yield`.

```
  bool isPrime = true;
  for (long possibleFactor = 2; possibleFactor <=
     (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)
  {
    long remainderAfterDivision = possiblePrime % possibleFactor;
```

```
      if (remainderAfterDivision == 0)
      {
        isPrime = false;
        break;
      }
    }
    if (isPrime)
    {
      yield return possiblePrime;
    }
  }
}
```

Если указаны слишком большие числа для нижнего и верхнего предела, в коде проявляется один интересный факт. При выполнении приложения результаты появляются по одному за раз, с паузами между ними, а не все одновременно. Это является свидетельством того, что код итератора возвращает результаты по одному, несмотря на тот факт, что никакого очевидного места завершения выполнения кода между вызовами `yield` нет. Однако "за кулисами" вызов `yield` все-таки действительно приводит к прерыванию выполнения кода, которое возобновляется при запросе следующего значения, т.е. тогда, когда у цикла `foreach`, в котором используется итератор, начинается новый прогон.

Итераторы и коллекции

Ранее в настоящей главе мы обещали, что объясним, каким образом можно применять итераторы для выполнения итерации по объектам, хранящимся в коллекции словарного типа, без привлечения объектов `DictionaryItem`. Вспомните, что класс коллекции `Animals` выглядит следующим образом:

```
public class Animals : DictionaryBase
{
  public void Add(string newID, Animal newAnimal)
  {
    Dictionary.Add(newID, newAnimal);
  }
  public void Remove(string animalID)
  {
    Dictionary.Remove(animalID);
  }
  public Animals()
  {
  }
  public Animal this[string animalID]
  {
    get
    {
      return (Animal)Dictionary[animalID];
    }
    set
    {
      Dictionary[animalID] = value;
    }
  }
}
```

Для получения желаемого поведения в этот код можно добавить простой итератор:

```
public new IEnumerator GetEnumerator()
{
  foreach (object animal in Dictionary.Values)
```

```
    yield return (Animal)animal;
}
```

Фрагмент кода ***DictionaryAnimals\Animals.cs***

Теперь для прохода по объектам `Animal` в коллекции можно использовать такой код:

```
foreach (Animal myAnimal in animalCollection)
{
  Console.WriteLine("New {0} object added to custom collection, " +
                "Name = {1}", myAnimal.ToString(), myAnimal.Name);
                // Новый объект {0} был добавлен в специальную коллекцию
}
```

Фрагмент кода ***DictionaryAnimals\Program.cs***

Этот код доступен в проекте `DictionaryAnimals`, который входит в состав кода примеров для настоящей главы.

Глубокое копирование

В главе 9 было показано, как провести поверхностное копирование за счет вызова защищенного метода `System.Object.MemberwiseClone()` внутри реализации `GetCopy()`:

```
public class Cloner
{
  public int Val;

  public Cloner(int newVal)
  {
    Val = newVal;
  }
  public object GetCopy()
  {
    return MemberwiseClone();
  }
}
```

Предположим, что есть поля, которые представляют собой не типы значения, а ссылочные типы (например, объекты):

```
public class Content
{
  public int Val;
}

public class Cloner
{
  public Content MyContent = new Content();

  public Cloner(int newVal)
  {
    MyContent.Val = newVal;
  }
  public object GetCopy()
  {
    return MemberwiseClone();
  }
}
```

В этом случае поверхностная копия, полученная методом `GetCopy()`, будет содержать поле, ссылающееся на тот же самый исходный объект. В следующем коде, использующем данный класс `Cloner`, иллюстрируются последствия выполнения в отношении ссылочных типов поверхностного копирования:

```
Cloner mySource = new Cloner(5);
Cloner myTarget = (Cloner)mySource.GetCopy();
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);
mySource.MyContent.Val = 2;
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);
```

В четвертой строке, в которой присваивается значение полю `mySource.MyContent.Val` (общедоступному полю `Val` общедоступного поля `MyContent` исходного объекта), также происходит изменение значения `myTarget.MyContent.Val`. Это объясняется тем, что `mySource.MyContent` ссылается на экземпляр того же самого объекта, что и `myTarget.MyContent`. Вывод в результате выполнения приведенного кода будет выглядеть так:

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 2
```

Обойти эту проблему можно за счет выполнения глубокого копирования. Можно соответствующим образом модифицировать приведенный ранее метод `GetCopy()`, однако предпочтительнее воспользоваться стандартным подходом .NET Framework: реализовать интерфейс `ICloneable`, который имеет единственный методом `Clone()`. Этот метод не принимает параметров и возвращает тип `object`, что делает его сигнатуру идентичной той, что была у применявшегося выше метода `GetCopy()`.

Попробуйте использовать следующий код глубокого копирования:

```
public class Content
{
  public int Val;
}

public class Cloner: ICloneable
{
  public Content MyContent = new Content();

  public Cloner(int newVal)
  {
    MyContent.Val = newVal;
  }

  public object Clone()
  {
    Cloner clonedCloner = new Cloner(MyContent.Val);
    return clonedCloner;
  }
}
```

В этом коде создается новый объект `Cloner` с применением поля `Val` объекта `Content`, содержащегося в исходном объекте `Cloner` (`MyContent`). Данное поле относится к типу значения, поэтому необходимости в выполнении более глубокого копирования нет.

Тестирование метода `Clone()` дает следующий результат:

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 5
```

На этот раз содержащиеся объекты являются независимыми. Обратите внимание, что иногда, в более сложных системах объектов, вызовы метода `Clone()` осуществляются рекурсивно. Например, если поле `MyContent` класса `Cloner` также требует глубокого копирования, нужный код может выглядеть следующим образом:

```
public class Cloner : ICloneable
{
  public Content MyContent = new Content();
  ...
```

```
  public object Clone()
  {
    Cloner clonedCloner = new Cloner();
    clonedCloner.MyContent = MyContent.Clone();
    return clonedCloner;
  }
}
```

Здесь для упрощения синтаксиса создания нового объекта `Cloner` вызывается конструктор по умолчанию. Чтобы этот код мог работать, понадобится также реализовать в классе `Content` интерфейс `ICloneable`.

Добавление возможности глубокого копирования в `CardLib`

Чтобы посмотреть на все это в деле, можно реализовать возможность копирования объектов `Card`, `Cards` и `Deck` с использованием интерфейса `ICloneable`. Это может быть полезно в некоторых карточных играх, где две колоды не должны ссылаться на один и тот же набор объектов `Card`, но порядок в одной колоде должен совпадать с порядком в другой.

Реализация функции клонирования для класса `Card` в `Ch11CardLib` выглядит довольно просто, поскольку для него вполне достаточно только возможности поверхностного копирования (т.к. в нем содержатся только данные типа значения, в виде полей). В определение этого класса понадобится внести следующие изменения:

```
public class Card : ICloneable
{
  public object Clone()
  {
    return MemberwiseClone();
  }
```

*Фрагмент кода **Ch11CardLib\Card.cs***

Эта реализация `ICloneable` предусматривает лишь поверхностное копирование. Никаких правил, указывающих, что должно происходить в методе `Clone()`, не предусмотрено, и для преследуемых здесь целей этого вполне достаточно.

Далее необходимо реализовать интерфейс `ICloneable` для класса коллекции `Cards`. Это будет немного сложнее, поскольку клонированию подлежит каждый объект `Card` в исходной коллекции, что, следовательно, требует применения глубокого копирования:

```
public class Cards : CollectionBase, ICloneable
{
  public object Clone()
  {
    Cards newCards = new Cards();
    foreach (Card sourceCard in List)
    {
      newCards.Add(sourceCard.Clone() as Card);
    }
    return newCards;
  }
```

*Фрагмент кода **Ch11CardLib\Cards.cs***

И, наконец, осталось реализовать интерфейс `ICloneable` для класса `Deck`. Здесь имеется одна небольшая проблема: дело в том, что помимо тасования в этом классе больше нет никаких возможностей для изменения содержащихся в нем карт. Например, в нем нельзя изменять экземпляр `Deck` для установки определенного порядка карт. Чтобы обойти эту проблему, потребуется определить для класса `Deck` новый приватный конструктор, кото-

рый позволит передавать конкретную коллекцию `Card` при создании экземпляра объекта `Deck`. Ниже показан код реализации клонирования для этого класса:

```
public class Deck : ICloneable
{
  public object Clone()
  {
    Deck newDeck = new Deck(cards.Clone() as Cards);
    return newDeck;
  }
  private Deck(Cards newCards)
  {
    cards = newCards;
  }
```

Фрагмент кода ***Ch11CardLib\Deck.cs***

Теперь можно протестировать все это с помощью простого клиентского кода (в методе `Main()`):

```
Deck deck1 = new Deck();
Deck deck2 = (Deck)deck1.Clone();
Console.WriteLine("The first card in the original deck is: {0}",
                  deck1.GetCard(0));
                // Первой картой в исходной колоде является: {0}
Console.WriteLine("The first card in the cloned deck is: {0}",
                  deck2.GetCard(0));
                // Первой картой в клонированной колоде является: {0}
deck1.Shuffle();
Console.WriteLine("Original deck shuffled.");
                // Исходная колода была перемешана
Console.WriteLine("The first card in the original deck is: {0}",
                  deck1.GetCard(0));
                // Первой картой в исходной колоде является: {0}
Console.WriteLine("The first card in the cloned deck is: {0}",
                  deck2.GetCard(0));
                // Первой картой в клонированной колоде является: {0}
Console.ReadKey();
```

Вывод будет выглядеть примерно так, как показано на рис. 11.5.

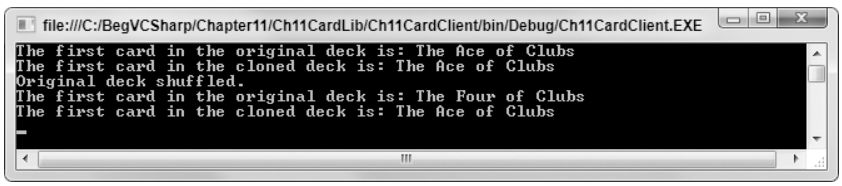


Рис. 11.5. *Тестирование глубокого копирования*

Сравнения

В этом разделе рассматриваются две разновидности сравнений между объектами:

- сравнение типов;
- сравнение значений.

Сравнения типов (т.е. определение того, что собой представляет объект или от чего он унаследован) играют важную роль во всех областях программирования на C#. Часто при передаче объекта — методу, например — происходящие дальше действия зависят от того,

к какому типу относится данный объект. Со сравнениями значений не раз приходилось встречаться при передаче объектов в этой и предшествующих главах, но здесь будут показаны дополнительные полезные приемы.

Сравнения значений тоже встречались неоднократно, по крайней мере, для простых типов. При сравнении значений объектов, однако, дело немного усложняется. Для начала необходимо определить, что понимается под сравнением и что операции вроде > означают в контексте классов. Это особенно важно в случае коллекций, для которых могут требоваться возможности сортировки объектов по какому-то условию, например, в алфавитном порядке или согласно более сложного алгоритма.

Сравнение типов

При сравнении объектов часто необходимо знать их тип, поскольку это позволяет выяснить, возможно ли сравнение их значений. В главе 9 был описан метод `GetType()`, который все классы наследуют от базового класса `System.Object`, и его применение вместе с операцией `typeof` для выяснения типа объекта (и выполнения в зависимости от этого того или иного действия):

```
if (myObj.GetType() == typeof(MyComplexClass))
{
  // myObj является экземпляром класса MyComplexClass.
}
```

Также было показано, как получить строковое представление типа объекта с помощью реализации по умолчанию метода `ToString()`, унаследованного от базового класса `System.Object`. Эти строковые представления впоследствии тоже могут сравниваться, но такой подход является довольно громоздким.

В настоящем разделе рассматривается более короткий и удобный способ для выяснения типа — использование операции `is`. Это позволяет получить более читабельный код, а также, как будет показано, просматривать базовые классы. Однако перед тем, как переходить к изучению операции `is`, необходимо ознакомиться с процессами, которые часто происходят "за кулисами" при работе с типами значения (в отличие от ссылочных типов): *упаковкой* и *распаковкой*.

Упаковка и распаковка

В главе 8 вы узнали об отличиях между ссылочными типами и типами значения, а в главе 9 эти отличия иллюстрировались на примере сравнения структур (типов значения) и классов (ссылочных типов). *Упаковкой* (boxing) называется процесс преобразования типа значения в тип `System.Object` или в тип интерфейса, который в нем реализован. *Распаковка* (unboxing) — это обратный процесс.

Например, предположим, что есть структура следующего типа:

```
struct MyStruct
{
  public int Val;
}
```

Упаковать структуру этого типа можно за счет ее помещения в переменную типа `object`:

```
MyStruct valType1 = new MyStruct();
valType1.Val = 5;
object refType = valType1;
```

Здесь создается новая переменная (по имени `valType1`) типа `MyStruct`, ее члену `Val` присваивается значение, после чего она упаковывается в переменную типа `object` (с именем `refType`).

В объекте, создаваемом такой упаковкой, будет содержаться ссылка на копию переменной valType, а не на исходную переменную valType. Удостовериться в этом можно, изменив содержимое исходной структуры, а затем распаковав содержащуюся в объекте структуру в новую переменную и просмотрев ее содержимое:

```
valType1.Val = 6;
MyStruct valType2 = (MyStruct)refType;
Console.WriteLine("valType2.Val = {0}", valType2.Val);
```

Выполнение этого кода приведет к получению следующего вывода:

```
valType2.Val = 5
```

В случае присваивания переменной типа object какого-то ссылочного типа поведение выглядит по-другому. Чтобы увидеть это, превратите MyStruct в класс (не обращая внимания на то, что имя класса получится не совсем подходящим):

```
class MyStruct
{
  public int Val;
}
```

Если не вносить изменений в приведенный ранее клиентский код (опять-таки, не обращая внимания на неподходящие имена для переменных), то в результате его выполнения будет получен следующий вывод:

```
valType2.Val = 6
```

Типы значения можно упаковывать в типы интерфейсов, при условии, разумеется, что эти интерфейсы в них реализованы. Например, предположим, что в типе MyStruct реализован интерфейс IMyInterface, как показано ниже:

```
interface IMyInterface
{
}

struct MyStruct : IMyInterface
{
 public int Val;
}
```

Тогда упаковать его в тип IMyInterface можно будет следующим образом:

```
MyStruct valType1 = new MyStruct();
IMyInterface refType = valType1;
```

Распаковка осуществляется с помощью обычного синтаксиса приведения:

```
MyStruct ValType2 = (MyStruct)refType;
```

Как видно по этим примерам, упаковка выполняется без вмешательства со стороны разработчика, т.е. писать какой-либо код для этого не требуется. Однако для распаковки значения требуется явное преобразование, которое обеспечивается приведением (упаковка выполняется неявно и потому такого требования не имеет).

Может возникнуть вопрос о том, когда конкретно необходимо делать нечто подобное? В действительности существуют две очень веских причины, по которым упаковка считается чрезвычайно полезной. Во-первых, она позволяет использовать типы значения в коллекциях (таких как ArrayList), где элементы имеют тип object. Во-вторых, она представляет собой внутренний механизм, который позволяет вызывать методы object на типах значения, таких как int и struct.

Напоследок следует также отметить, что распаковка необходима, чтобы стал возможным доступ к содержимому типа значения.

Операция `is`

Операция `is` — это не способ определения того, *относится* ли объект к конкретному типу. Вместо этого она позволяет проверить, *является или может ли объект быть преобразован в* заданный тип. Если это так, операция `is` возвращает `true`.

В более ранних примерах применялись классы `Cow` и `Chicken`, оба унаследованные от класса `Animal`. Операция `is` для сравнения объектов с типом `Animal` будет давать `true` для объектов всех трех типов, а не только для `Animal`. Добиться подобного поведения с помощью показанных ранее метода `GetType()` и операции `typeof()` было бы нелегко.

Синтаксис операции `is` выглядит следующим образом:

`<операнд> is <тип>`

Ниже описаны результаты этого выражения.

- Если `<тип>` является типом класса, `true` возвращается в случае, когда `<операнд>` относится к этому типу, унаследован от этого типа или может быть упакован в этот тип.
- Если `<тип>` является типом интерфейса, `true` возвращается в случае, когда `<операнд>` относится к этому типу или к типу, который реализует интерфейс.
- Если `<тип>` является типом значения, `true` возвращается в случае, когда `<операнд>` относится к этому типу или к типу, который может быть распакован в этот тип.

Следующее практическое занятия посвящено применению операции `is`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование операции `is`

1. Создайте новое консольное приложение по имени `Ch11Ex04` и сохраните его в каталоге `C:\BegVCSharp\Chapter11`.
2. Измените код в его файле `Program.cs` следующим образом:

```
namespace Ch11Ex04
{
  class Checker
  {
    public void Check(object param1)
    {
      if (param1 is ClassA)
        Console.WriteLine("Variable can be converted to ClassA.");
                    // Переменная может быть преобразована в ClassA
      else
        Console.WriteLine("Variable can't be converted to ClassA.");
                    // Переменная не может быть преобразована в ClassA

      if (param1 is IMyInterface)
        Console.WriteLine("Variable can be converted to IMyInterface.");
                    // Переменная может быть преобразована в IMyInterface
      else
        Console.WriteLine("Variable can't be converted to IMyInterface.");
                    // Переменная не может быть преобразована в IMyInterface

      if (param1 is MyStruct)
        Console.WriteLine("Variable can be converted to MyStruct.");
                    // Переменная может быть преобразована в MyStruct
      else
        Console.WriteLine("Variable can't be converted to MyStruct.");
                    // Переменная не может быть преобразована в MyStruct
    }
  }
```

```
interface IMyInterface
{
}
class ClassA : IMyInterface
{
}
class ClassB : IMyInterface
{
}
class ClassC
{
}
class ClassD : ClassA
{
}
struct MyStruct : IMyInterface
{
}
class Program
{
  static void Main(string[] args)
  {
    Checker check = new Checker();
    ClassA try1 = new ClassA();
    ClassB try2 = new ClassB();
    ClassC try3 = new ClassC();
    ClassD try4 = new ClassD();
    MyStruct try5 = new MyStruct();
    object try6 = try5;
    Console.WriteLine("Analyzing ClassA type variable:");
                    // Анализ переменной типа ClassA
    check.Check(try1);
    Console.WriteLine("\nAnalyzing ClassB type variable:");
                    // Анализ переменной типа ClassB
    check.Check(try2);
    Console.WriteLine("\nAnalyzing ClassC type variable:");
                    // Анализ переменной типа ClassC
    check.Check(try3);
    Console.WriteLine("\nAnalyzing ClassD type variable:");
                    // Анализ переменной типа ClassD
    check.Check(try4);
    Console.WriteLine("\nAnalyzing MyStruct type variable:");
                    // Анализ переменной типа MyStruct
    check.Check(try5);
    Console.WriteLine("\nAnalyzing boxed MyStruct type variable:");
                    // Анализ упакованной переменной типа MyStruct
    check.Check(try6);
    Console.ReadKey();
  }
 }
}
```

Фрагмент кода ***Ch11Ex04\Program.cs***

3. Запустите приложение. На рис. 11.6 показан результат, который должен получиться.

Описание работы

В этом примере иллюстрируются разнообразные результаты, которые можно получать, применяя операции `is`. Сначала определяются три класса, интерфейс и структура, которые затем передаются в качестве параметров методу класса. В этом методе с помощью опе-

рации `is` определяется возможность их преобразования в тип `ClassA`, тип интерфейса и тип структуры.

Только типы `ClassA` и `ClassD` (унаследованный от `ClassA`) являются совместимыми с `ClassA`. Типы, которые не наследуются от этого класса, соответственно, не совместимы с ним.

В типах `ClassA`, `ClassB` и `MyStruct` реализован интерфейс `IMyInterface` и потому все они являются совместимыми с типом `IMyInterface`. Тип `ClassD` унаследован от `ClassA` и, следовательно, совместим с этим типом. Таким образом, несовместимым оказывается только тип `ClassC`.

И, наконец, совместимыми с `MyStruct` являются только переменные самого типа `MyStruct` и переменные, упакованные в этот тип, поскольку нельзя преобразовать ссылочные типы в типы значения (хотя, разумеется, можно распаковывать ранее упакованные переменные).

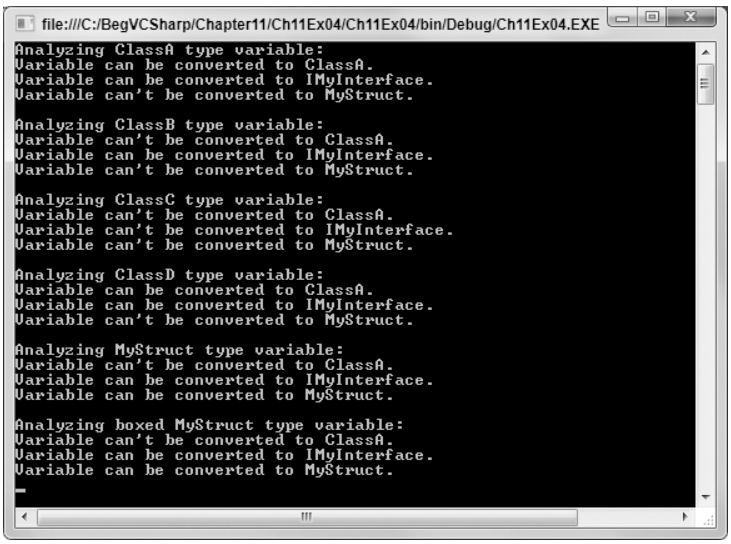


Рис. 11.6. *Приложение `Ch11Ex04` в действии*

Сравнение значений

Рассмотрим два представляющих людей объекта `Person`, которые имеют целочисленное свойство `Age` (Возраст). Может понадобиться сравнить их для того, чтобы узнать, кто из двух людей старше. Для этого подойдет следующий код:

```
if (person1.Age > person2.Age)
{
  ...
}
```

Такой подход хорош, но доступны и альтернативные варианты. Например, может оказаться предпочтительнее использовать код, показанный ниже:

```
if (person1 > person2)
{
  ...
}
```

Подобный синтаксис возможен благодаря *перегрузке операций*, которая рассматривается в следующем разделе. Это мощный прием, но применять его нужно осмотрительно. По

приведенному выше коду нельзя сразу сказать, что сравнивается именно возраст людей, а не рост, вес, коэффициент умственного развития или вообще общая "значимость".

Еще одним вариантом является использование интерфейсов `IComparable` и `IComparer`, которые позволяют определять то, как объекты должны сравниваться друг с другом, стандартным образом. Применение такого приема поддерживается во многих классах коллекций в .NET Framework, что делает его замечательным способом для проведения сортировки объектов в коллекции.

Перегрузка операций

Перегрузка операций (operator overloading) позволяет использовать стандартные операции, такие как +, > и т.д., в классах собственной разработки. "Перегрузкой" этот прием называется потому, что предусматривает предоставление для этих операций собственных реализаций, когда операции используются с параметрами специфических типов. Это во многом похоже на перегрузку методов, при которой методам с одинаковым именем передаются разные параметры.

Механизм перегрузки операций позволяет предусмотреть любую обработку в реализации перегружаемой операции. При этом действия могут быть намного сложнее, чем, например, сложение двух операндов в случае операции +. Позже, во время дальнейшей модернизации библиотеки `CardLib`, это будет продемонстрировано на примере реализации операции сравнения двух карт, которая позволяет выяснить, какая из них бьет другую в данной партии.

Поскольку выигрыш партии во многих карточных играх зависит от масти участвующих карт, простым сравнением чисел на картах здесь не обойтись. Если вторая выкладываемая карта по масти отличается от первой, тогда должна побеждать первая карта, независимо от ее достоинства. Реализовать такое поведение можно за счет принятия во внимание порядка следования двух операндов. Еще можно позаботиться о том, чтобы во внимание принималась козырная масть и тогда козырная карта могла бы бить карты любых других мастей, даже если она не выкладывается первой. Это означает, что результат вычисления `card1 > card2`, равный `true` (т.е. `card1` бьет `card2`, если `card1` выкладывается первой), не обязательно подразумевает, что результатом вычисления `card2 > card1` будет `false`. В случае если ни `card1`, ни `card2` не являются козырными картами и имеют разную масть, оба этих выражения должны возвращать `true`.

Сначала давайте посмотрим на базовый синтаксис перегрузки операций. Операции можно перегружать добавлением в класс членов `operator` (которые обязательно должны быть `static`). Некоторые операции обладают несколькими способами применения (как, например, операция –, у которой имеется унарная и бинарная версии), поэтому необходимо указать количество и типы операндов. Как правило, типы операндов совпадают с типом класса, в котором определяется операция, хотя допустимо также определять операции, способные работать со смешанными типами (это будет показано позже).

В качестве примера рассмотрим простой тип `AddClass1` со следующим определением:

```
public class AddClass1
{
  public int val;
}
```

Это просто оболочка вокруг значения `int`, но с ее помощью можно проиллюстрировать базовые принципы. Имея такой класс, приведенный ниже код компилироваться не будет:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
AddClass1 op3 = op1 + op2;
```

Сообщение об ошибке будет гласить, что операция + не может применяться к операндам типа AddClass1. Причина в том, что подлежащая выполнению операция пока еще не определена. Показанный ниже код будет работать, но не обеспечит ожидаемый результат:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
bool op3 = op1 == op2;
```

Здесь op1 и op2 сравниваются с применением бинарной операции ==, которая позволяет выяснить, ссылаются ли они на один и тот же объект, а *не* для проверки, равны ли их значения между собой. op3 возвращает в предыдущем коде значение false, даже несмотря на то, что op1.val и op2.val идентичны.

Для перегрузки операции + можно использовать такой код:

```
public class AddClass1
{
  public int val;

  public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
  {
    AddClass1 returnVal = new AddClass1();
    returnVal.val = op1.val + op2.val;
    return returnVal;
  }
}
```

Как здесь видно, перегрузки операций выглядят во многом подобно стандартным объявлениям статических методов, но только в них используется ключевое слово operator и сама операция, а не имя метода. Теперь операцию + можно успешно использовать с данным классом:

```
AddClass1 op3 = op1 + op2;
```

Перегрузка всех бинарных операций осуществляется по одной и той же схеме. Унарные операции выглядят похоже, но принимают только один параметр:

```
public class AddClass1
{
  public int val;
  public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
  {
    AddClass1 returnVal = new AddClass1();
    returnVal.val = op1.val + op2.val;
    return returnVal;
  }
  public static AddClass1 operator -(AddClass1 op1)
  {
    AddClass1 returnVal = new AddClass1();
    returnVal.val = -op1.val;
    return returnVal;
  }
}
```

Обе операции работают с операндами того же типа, что и у класса, и имеют возвращаемые значения, тип которых тоже совпадает с типом данного класса. Однако давайте рассмотрим классы со следующими определениями:

```
public class AddClass1
{
  public int val;
```

```
  public static AddClass3 operator +(AddClass1 op1, AddClass2 op2)
  {
    AddClass3 returnVal = new AddClass3();
    returnVal.val = op1.val + op2.val;
    return returnVal;
  }
}
public class AddClass2
{
  public int val;
}
public class AddClass3
{
  public int val;
}
```

Они позволят использовать такой код:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass2 op2 = new AddClass2();
op2.val = 5;
AddClass3 op3 = op1 + op2;
```

При необходимости типы можно смешивать подобным образом. Однако обратите внимание, что если добавить такую же операцию в `AddClass2`, то предыдущий код работать не будет, поскольку появится неоднозначность относительно выбора операции для использования. Это означает, что следует соблюдать осторожность и не добавлять операции с одинаковой сигнатурой в более чем один класс.

Вдобавок, в случае смешивания типов, операнды *должны* предоставляться в том же порядке, что и параметры в перегруженной операции. В случае неверного порядка операндов операция работать не будет. Например, использовать следующую операцию нельзя:

```
AddClass3 op3 = op2 + op1;
```

Чтобы она работала, потребуется предоставить еще одну перегруженную версию с параметрами, идущими в обратном порядке:

```
public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)
{
  AddClass3 returnVal = new AddClass3();
  returnVal.val = op1.val + op2.val;
  return returnVal;
}
```

Ниже перечислены операции, которые могут быть перегружены.

- **Унарные операции**: +, –, !, ~, ++, -- , `true`, `false`
- **Бинарные операции**: +, – , *, /, %, & , |, ^, < <, > >
- **Операции сравнения**: ==, !=, < , > , <=, >=

В случае перегрузки операций `true` и `false` можно будет использовать соответствующие классы в булевских выражениях, наподобие `if (op1) {}`.

Перегружать операции присваивания вроде += не разрешено, однако у таких операций имеются простые эквиваленты, такие как +, поэтому беспокоиться о них не стоит. Перегрузка операции + означает, что и операция += будет функционировать должным образом. Перегружать операцию = не допускается из-за ее фундаментального предназначения, но с ней связаны определяемые пользователем операции преобразования, которые будут рассматриваться в следующем разделе.

Кроме того, нельзя перегружать и такие операции, как && и ||, но внутри них для выполнения вычислений применяются операции & и |, поэтому вполне достаточно перегрузить их.

Некоторые операции, например, < и >, должны перегружаться парами. То есть перегрузить операцию < и не перегружать при этом операцию > не допускается. Во многих случаях внутри реализации можно просто вызвать другую операцию, сократив объем необходимого кода (и, следовательно, ошибок, которые могут возникать):

```
public class AddClass1
{
  public int val;
  public static bool operator >=(AddClass1 op1, AddClass1 op2)
  {
    return (op1.val >= op2.val);
  }
  public static bool operator < (AddClass1 op1, AddClass1 op2)
  {
    return !(op1 >= op2);
  }
  // Также необходимы реализации операций <= и >.
}
```

В определениях более сложных операций такой подход действительно помогает сократить количество строк кода, и, следовательно, объем изменяемого кода, если в будущем возникнет необходимость корректировки реализации этих операций.

То же самое касается операций == и !=, но в их случае часто лучше переопределять методы `Object.Equals()` и `Object.GetHashCode()`, поскольку они также могут использоваться для сравнения объектов. Переопределение этих методов гарантирует, что какой бы прием не применялся пользователями класса, будет получен один и тот же результат. Существенным это не является, но заслуживает упоминания для полноты изложения и требует использования следующих нестатических переопределенных методов:

```
public class AddClass1
{
  public int val;
  public static bool operator ==(AddClass1 op1, AddClass1 op2)
  {
    return (op1.val == op2.val);
  }
  public static bool operator !=(AddClass1 op1, AddClass1 op2)
  {
    return !(op1 == op2);
  }
  public override bool Equals(object op1)
  {
    return val == ((AddClass1)op1).val;
  }
  public override int GetHashCode()
  {
    return val;
  }
}
```

Метод `GetHashCode()` служит для получения уникального значения `int` для экземпляра объекта на основе его состояния. Использование `val` здесь является допустимым, потому что `val` тоже является значением типа `int`.

Обратите внимание, что в методе `Equals()` применяется параметр типа `object`. Такая сигнатура является обязательной, иначе это будет перегрузка, а не переопределение данного метода, и тогда его реализация по умолчанию все равно будет доступна пользовате-

лям класса. Поэтому в нем для получения необходимого результата должно использоваться приведение типов. Часто полезно еще и проверять тип объекта с помощью рассмотренной ранее операции `is`, как показано ниже:

```
public override bool Equals(object op1)
{
  if (op1 is AddClass1)
  {
    return val == ((AddClass1)op1).val;
  }
  else
  {
    throw new ArgumentException(
      "Cannot compare AddClass1 objects with objects of type "
      + op1.GetType().ToString());
  }
}
```

В этом коде исключение генерируется в случае, если передаваемый методу `Equals` операнд относится не к тому типу, а его преобразование к правильному типу невозможно. Конечно, такое поведение может оказаться не тем, что нужно. Может потребоваться возможность сравнения объектов одного типа с объектами другого типа; в этом случае необходимо дополнительное ветвление. Или же может понадобиться ограничить сравнение объектами в точности одинакового типа, что требует внесения следующего изменения в первый оператор `if`:

```
if (op1.GetType() == typeof(AddClass1))
```

Добавление перегруженных операций в `CardLib`

В этом разделе мы снова обновим проект `Ch11CardLib`, перегрузив в классе `Card` ряд операций. Однако сначала необходимо добавить в класс `Card` дополнительные поля, позволяющие определять козырные масти и назначать тузы старшей картой. Эти поля должны быть статическими, поскольку установленные в них значения будут применяться ко всем объектам `Card`.

```
public class Card
{
  /// <summary>
  /// Флаг для применения козырных карт. При значении true козырные
  /// карты должны цениться больше, чем карты других мастей.
  /// </summary>
  public static bool useTrumps = false;

  /// <summary>
  /// Козырная масть, которая должна использоваться в случае,
  /// если useTrumps равно true.
  /// </summary>
  public static Suit trump = Suit.Club;

  /// <summary>
  /// Флаг, определяющий то, являются ли тузы старше королей или младше двоек.
  /// </summary>
  public static bool isAceHigh = true;
```

Фрагмент кода `Ch11CardLib\Card.cs`

Эти правила применяются ко всем объектам `Card` в каждом экземпляре `Deck` приложения. Нельзя иметь две колоды карт, подчиняющиеся разным правилам. Однако для данной библиотеки подобное поведение является допустимым, т.к. можно предположить, что если приложению нужны отдельные правила, оно может поддерживать их самостоятельно, например, за счет установки этих статических членов `Card` при каждой смене колоды.

Теперь добавим в класс Deck несколько конструкторов для инициализации колод с разными характеристиками:

```
/// <summary>
/// Конструктор не по умолчанию. Позволяет назначать тузы старшими.
/// </summary>
public Deck(bool isAceHigh) : this()
{
  Card.isAceHigh = isAceHigh;
}

/// <summary>
/// Конструктор не по умолчанию. Позволяет использовать козырную масть.
/// </summary>
public Deck(bool useTrumps, Suit trump) : this()
{
  Card.useTrumps = useTrumps;
  Card.trump = trump;
}

/// <summary>
// Конструктор не по умолчанию. Позволяет назначать тузы старшими
// и использовать козырную масть.
/// </summary>
public Deck(bool isAceHigh, bool useTrumps, Suit trump) : this()
{
  Card.isAceHigh = isAceHigh;
  Card.useTrumps = useTrumps;
  Card.trump = trump;
}
```

Фрагмент кода ***Ch11CardLib\Deck.cs***

Каждый из этих конструкторов определяется с использованием показывавшегося в главе 9 синтаксиса : this(), чтобы во всех случаях перед конструктором не по умолчанию вызывался конструктор по умолчанию и инициализировал колоду.

Теперь можно добавить в класс Card перегруженные операции:

```
public static bool operator ==(Card card1, Card card2)
{
  return (card1.suit == card2.suit) && (card1.rank == card2.rank);
}
public static bool operator !=(Card card1, Card card2)
{
  return !(card1 == card2);
}
public override bool Equals(object card)
{
  return this == (Card)card;
}
public override int GetHashCode()
{
  return 13*(int)rank + (int)suit;
}
public static bool operator >(Card card1, Card card2)
{
  if (card1.suit == card2.suit)
  {
    if (isAceHigh)
    {
      if (card1.rank == Rank.Ace)
      {
```

```
        if (card2.rank == Rank.Ace)
          return false;
        else
          return true;
      }
      else
      {
        if (card2.rank == Rank.Ace)
           return false;
        else
         return (card1.rank > card2.rank);
      }
    }
    else
    {
      return (card1.rank > card2.rank);
    }
  }
  else
  {
    if (useTrumps && (card2.suit == Card.trump))
      return false;
    else
      return true;
  }
}
public static bool operator <(Card card1, Card card2)
{
  return !(card1 >= card2);
}
public static bool operator >=(Card card1, Card card2)
{
  if (card1.suit == card2.suit)
  {
    if (isAceHigh)
    {
      if (card1.rank == Rank.Ace)
      {
        return true;
      }
      else
      {
        if (card2.rank == Rank.Ace)
          return false;
        else
          return (card1.rank >= card2.rank);
      }
    }
    else
    {
      return (card1.rank >= card2.rank);
    }
  }
  else
  {
    if (useTrumps && (card2.suit == Card.trump))
      return false;
    else
      return true;
  }
}
```

```
public static bool operator <=(Card card1, Card card2)
{
  return !(card1 > card2);
}
```

Фрагмент кода ***Ch11CardLib\Card.cs***

В этом коде нет ничего примечательного, за исключением разве что относительно длинного кода для перегрузки операций > и >=. Пошагово пройдя по коду операции >, можно легко понять, как он работает.

Здесь выполняется сравнение двух карт — `card1` и `card2`, — при котором предполагается, что карта `card1` была выложена на стол первой. Как уже объяснялось ранее, это важно в случае использования козырных карт, поскольку козырная карта будет бить не козырную, даже если та является картой более высокого достоинства. Конечно, если масть обеих карт совпадает, тогда то, является масть козырной или нет, роли не играет, поэтому первым выполняется такое сравнение:

```
public static bool operator > (Card card1, Card card2)
{
  if (card1.suit == card2.suit)
  {
```

Если статический флаг `isAceHigh` установлен в `true`, сравнивать достоинства карт напрямую по их значению в перечислении `Rank` нельзя, т.к. достоинство туза соответствует в этом перечислении значению `1`, которое меньше значений карт остальных достоинств. Вместо этого используются описанные ниже шаги.

- Если первой картой является туз, тогда выполняется проверка, является ли вторая карта тоже тузом. Если да, тогда первая карта не бьет вторую, а если нет, тогда побеждает первая карта:

```
if (isAceHigh)
{
  if (card1.rank == Rank.Ace)
  {
    if (card2.rank == Rank.Ace)
      return false;
    else
      return true;
  }
```

- Если первая карта отлична от туза, тогда все равно выполняется проверка, не является ли тузом вторая карта. Если да, тогда побеждает вторая карта, а если нет, тогда может выполняться сравнение значений, представляющих достоинство карт, поскольку уже известно, что тузов среди них нет:

```
  else
  {
    if (card2.rank == Rank.Ace)
      return false;
    else
      return (card1.rank > card2.rank);
  }
}
```

- Если тузы не являются старшей картой, тогда сразу же производится сравнение значений, представляющих достоинство карт:

```
else
{
   return (card1.rank > card2.rank);
}
```

В остальном коде обрабатывается ситуация, когда масти `card1` и `card2` не совпадают. Здесь важную роль играет статический флаг `useTrumps`. Если его значение рано `true`, а масть карты `card2` — козырная, тогда можно точно сказать, что карта `card1` козырем не является (поскольку масти этих двух карт не совпадают), а козыри всегда побеждают, поэтому карта `card2` должна быть признана старшей:

```
    else
    {
      if (useTrumps && (card2.suit == Card.trump))
        return false;
```

Если карта `card2` не является козырем (или значение флага `useTrumps` равно `false`), тогда должна побеждать карта `card1`, поскольку она была выложена на стол первой:

```
      else
        return true;
    }
}
```

Только в еще одной операции (>=) используется подобный сложный код; все остальные операции выглядят очень просто, поэтому рассказывать о них более подробно нет никакой необходимости.

Ниже приведен простой клиентский код, который можно использовать для тестирования всех этих операций (и который следует поместить в метод `Main()`, подобно тому, как это делалось с клиентским кодом в предыдущих примерах `CardLib`):

```
Card.isAceHigh = true;
Console.WriteLine("Aces are high.");
                  // Тузы являются старшей картой
Card.useTrumps = true;
Card.trump = Suit.Club;
Console.WriteLine("Clubs are trumps.");
                  // Трефы являются козырной мастью
Card card1, card2, card3, card4, card5;
card1 = new Card(Suit.Club, Rank.Five);
card2 = new Card(Suit.Club, Rank.Five);
card3 = new Card(Suit.Club, Rank.Ace);
card4 = new Card(Suit.Heart, Rank.Ten);
card5 = new Card(Suit.Diamond, Rank.Ace);
Console.WriteLine("{0} == {1} ? {2}",
card1.ToString(), card2.ToString(), card1 == card2);
Console.WriteLine("{0} != {1} ? {2}",
   card1.ToString(), card3.ToString(), card1 != card3);
Console.WriteLine("{0}.Equals({1}) ? {2}",
   card1.ToString(), card4.ToString(), card1.Equals(card4));
Console.WriteLine("Card.Equals({0}, {1}) ? {2}",
   card3.ToString(), card4.ToString(), Card.Equals(card3, card4));
Console.WriteLine("{0} > {1} ? {2}",
   card1.ToString(), card2.ToString(), card1 > card2);
Console.WriteLine("{0} <= {1} ? {2}",
   card1.ToString(), card3.ToString(), card1 <= card3);
Console.WriteLine("{0} > {1} ? {2}",
   card1.ToString(), card4.ToString(), card1 > card4);
Console.WriteLine("{0} > {1} ? {2}",
   card4.ToString(), card1.ToString(), card4 > card1);
Console.WriteLine("{0} > {1} ? {2}",
   card5.ToString(), card4.ToString(), card5 > card4);
Console.WriteLine("{0} > {1} ? {2}",
   card4.ToString(), card5.ToString(), card4 > card5);
Console.ReadKey();
```

Фрагмент кода ***Ch11CardClient\Program.cs***

На рис. 11.7 показан результат выполнения этого кода.

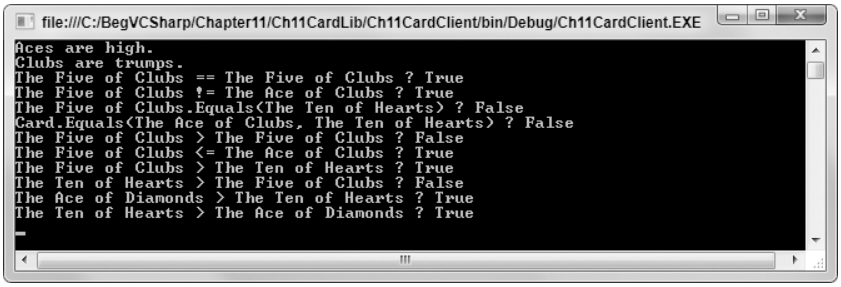


Рис. 11.7. *Тестирование перегруженных операций в* `CardLib`

В каждом случае операции применяются с учетом заданных правил. Это особенно заметно в четырех последних строках вывода, демонстрирующих, что козырные карты всегда бьют не козырные.

Интерфейсы `IComparable` и `IComparer`

Интерфейсы `IComparable` и `IComparer` представляют собой стандартный способ для сравнения объектов в .NET Framework. Разница между ними состоит в следующем.

- Интерфейс `IComparable` реализуется в классе объекта, который подлежит сравнению, и потому позволяет выполнять сравнения только между этим и еще каким-нибудь объектом.
- Интерфейс `IComparer` реализуется в отдельном классе и потому позволяет выполнять сравнения между двумя любыми объектами.

Обычно классу предоставляется код сравнения по умолчанию за счет применения `IComparable` и код сравнений не по умолчанию, использующий другие классы.

Интерфейс `IComparable` имеет единственный метод `CompareTo()`, который принимает в качестве параметра любой объект. Например, его можно реализовать так, чтобы он принимал объект `Person`, представляющий человека, и определял, является ли данный человек старше или младше текущего. В действительности этот метод возвращает значение `int`, так что можно еще и определить, насколько второй человек старше или моложе.

```
if (person1.CompareTo(person2) == 0)
{
  Console.WriteLine("Same age");
                  // Того же возраста
}
else if (person1.CompareTo(person2) > 0)
{
  Console.WriteLine("person 1 is Older");
                  // Первый человек старше
}
else
{
  Console.WriteLine("person1 is Younger");
                  // Первый человек моложе
}
```

Интерфейс `IComparer` тоже имеет только один метод `Compare()`, который принимает в качестве параметров два объекта и, подобно методу `CompareTo()`, возвращает целочисленный результат. При наличии объекта, поддерживающего `IComparer`, можно использовать код, подобный показанному ниже:

```
if (personComparer.Compare(person1, person2) == 0)
{
  Console.WriteLine("Same age");
                  // Того же возраста
}
else if (personComparer.Compare(person1, person2) > 0)
{
  Console.WriteLine("person 1 is Older");
                  // Первый человек старше
}
else
{
  Console.WriteLine("person1 is Younger");
                  // Первый человек моложе
}
```

В обоих случаях передаваемые методам параметры относятся к типу `System.Object`. Это означает, что сравнивать между собой можно объекты любого типа, из-за чего перед возвращением результата обычно требуется выполнить сравнение типов с возможной генерацией исключения, если используются недопустимые типы.

В .NET Framework класс `Comparer` из пространства имен `System.Collections` предлагает реализацию по умолчанию интерфейса `IComparer`. Этот класс способен выполнять специфичные для культуры операции сравнения между простыми типами, а также любыми типами, которые поддерживают интерфейс `IComparable`. Его можно использовать, например, следующим образом:

```
string firstString = "First String";
string secondString = "Second String";
Console.WriteLine("Comparing '{0}' and '{1}', result: {2}",
                  // Результат сравнения строк
   firstString, secondString,
   Comparer.Default.Compare(firstString, secondString));
int firstNumber = 35;
int secondNumber = 23;
Console.WriteLine("Comparing '{0}' and '{1}', result: {2}",
                  // Результат сравнения чисел
   firstNumber, secondNumber,
   Comparer.Default.Compare(firstNumber, secondNumber));
```

В коде сначала используется статический член `Comparer.Default` для получения экземпляра класса `Comparer`, после чего с помощью метода `Compare()` сравниваются первые две строки и затем два числа.

Результат получается следующий:

```
Comparing 'First String' and 'Second String', result: -1
Comparing '35' and '23', result: 1
```

Поскольку в алфавитном порядке буква F находится перед буквой S, получается, что буква F является "меньше" буквы S и поэтому в результате первого сравнения возвращается значение `-1`. Аналогично, число 35 больше числа 23 и потому в этом случае в результате возвращается значение `1`. Обратите внимание, что величина разницы в результатах не отражается.

Класс `Comparer` должен использоваться только с типами, которые могут сравниваться. При попытке сравнить `firstString` с `firstNumber`, например, будет генерироваться исключение. Ниже перечислен еще ряд моментов, связанных с поведением этого класса.

- Объекты, передаваемые `Comparer.Compare()`, проверяются на предмет поддержки интерфейса `IComparable`. Если поддерживают, используется его реализация.
- Нулевые значения являются допустимыми и считаются "меньше" любого другого объекта.

- Строки обрабатываются в соответствии с текущей культурой. Чтобы обработать строки в соответствии с другой культурой (или языком), экземпляр класса `Comparer` должен быть создан с помощью своего конструктора, который позволяет передавать объект `System.Globalization.CultureInfo`, указывающий на применяемую культуру.
- Строки обрабатываются с учетом регистра символов. Чтобы они обрабатывались без учета регистра, необходимо использовать класс `CaseInsensitiveComparer`, который во всем остальном работает совершенно аналогично.

Сортировка коллекций с использованием интерфейсов `IComparable` и `IComparer`

Многие классы коллекций допускают сортировку либо за счет применения операций сравнения по умолчанию между объектами, либо с использованием специальных методов. Одним из примеров является класс `ArrayList`. В нем содержится метод `Sort()`, который может вызываться без параметров, в случае чего в нем используются операции сравнения по умолчанию, а также с параметром в виде интерфейса `IComparer`, который применяется для сравнения пар объектов.

Когда коллекция `ArrayList` заполнена простыми типами, такими как целые числа или строки, сравнения по умолчанию вполне подходят. В собственных классах потребуется либо реализовать интерфейс `IComparable`, либо создать отдельный класс, поддерживающий интерфейс `IComparer`, и применять его для выполнения сравнений.

Обратите внимание, что некоторые классы в пространстве имен `System.Collection`, в том числе `CollectionBase`, не имеют метода для сортировки. Сортировка коллекции, унаследованной от этого класса, потребует немного большей работы, связанной с реализацией сортировки внутренней коллекции `List`.

В следующем практическом занятии демонстрируется использование стандартного и нестандартного методов сравнения для выполнения сортировки списка.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Сортировка списка

1. Создайте новое консольное приложение по имени `Ch11Ex05` и сохраните его в каталоге `C:\BegVCSharp\Chapter11`.
2. Добавьте новый класс `Person` и модифицируйте его код следующим образом:

```
namespace Ch11Ex05
{
  class Person : IComparable
  {
    public string Name;
    public int Age;

    public Person(string name, int age)
    {
      Name = name;
      Age = age;
    }

    public int CompareTo(object obj)
    {
      if (obj is Person)
      {
        Person otherPerson = obj as Person;
        return this.Age - otherPerson.Age;
      }
```

```
      else
      {
        throw new ArgumentException(
          "Object to compare to is not a Person object.");
          // Объект, подлежащий сравнению, не является объектом Person
      }
    }
  }
}
```

*Фрагмент кода **Ch11Ex05\Person.cs***

3. Добавьте еще один новый класс по имени `PersonComparerName` и модифицируйте его, как показано ниже:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex05
{
  public class PersonComparerName : IComparer
  {
   public static IComparer Default = new PersonComparerName();
   public int Compare(object x, object y)
   {
     if (x is Person && y is Person)
     {
       return Comparer.Default.Compare(
        ((Person)x).Name, ((Person)y).Name);
     }
     else
     {
       throw new ArgumentException(
        "One or both objects to compare are not Person objects.");
        // Один или оба сравниваемых объекта
        // не являются объектами Person
     }
   }
  }
}
```

*Фрагмент кода **Ch11Ex05\PersonComparerName.cs***

4. Измените код в файле `Program.cs` следующим образом:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex05
{
  class Program
  {
    static void Main(string[] args)
    {
      ArrayList list = new ArrayList();
      list.Add(new Person("Jim", 30));
      list.Add(new Person("Bob", 25));
      list.Add(new Person("Bert", 27));
      list.Add(new Person("Ernie", 22));
```

```
            Console.WriteLine("Unsorted people:");
                            // Несортированный список людей
            for (int i = 0; i < list.Count; i++)
            {
              Console.WriteLine("{0} ({1})",
                 (list[i] as Person).Name, (list[i] as Person).Age);
            }
            Console.WriteLine();

            Console.WriteLine(
                "People sorted with default comparer (by age):");
              // Список людей, отсортированный (по возрасту)
              // с применением стандартного метода сравнения

            list.Sort();
            for (int i = 0; i < list.Count; i++)
            {
              Console.WriteLine("{0} ({1})",
               (list[i] as Person).Name, (list[i] as Person).Age);
            }
            Console.WriteLine();

            Console.WriteLine(
                "People sorted with nondefault comparer (by name):");
              // Список людей, отсортированный (по имени) с применением
              // нестандартного метода сравнения

            list.Sort(PersonComparerName.Default);
            for (int i = 0; i < list.Count; i++)
            {
              Console.WriteLine("{0} ({1})",
                 (list[i] as Person).Name, (list[i] as Person).Age);
            }

            Console.ReadKey();
          }
        }
    }
```

Фрагмент кода ***Ch11Ex05\Program.cs***

5. Запустите приложение. На рис. 11.8 показан результат, который должен получиться.

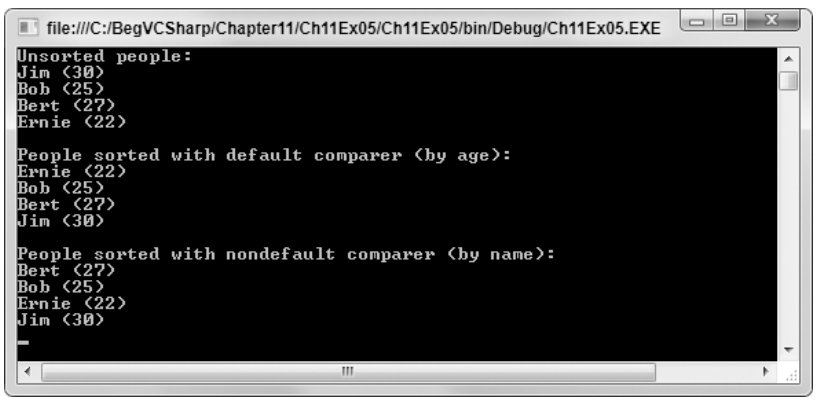


Рис. 11.8. *Приложение* `Ch11Ex05` *в действии*

Описание работы

Здесь класс коллекции `ArrayList`, содержащий объекты `Person`, сортируются двумя разными способами. Вызовом метода `ArrayList.Sort()` применяется стандартный метод сравнения, которым является метод `CompareTo()` класса `Person` (поскольку в этом классе реализован интерфейс `IComparable`):

```
public int CompareTo(object obj)
{
  if (obj is Person)
  {
    Person otherPerson = obj as Person;
    return this.Age - otherPerson.Age;
  }
  else
  {
    throw new ArgumentException(
     "Object to compare to is not a Person object.");
     // Объект, подлежащий сравнению, не является объектом Person
  }
}
```

В этом методе сначала производится проверка, может ли переданный в аргументе объект сравниваться с объектом `Person`, т.е. может ли он быть преобразован в `Person`. Если нет, генерируется исключение. Если да, тогда производится сравнение значений свойств `Age` этих двух объектов `Person`.

Далее выполняется сортировка посредством нестандартного метода сравнения с использованием класса `PersonComparerName`, в котором реализован интерфейс `IComparer`. Для упрощения работы с ним данный класс имеет поле `public static`:

```
public static IComparer Default = new PersonComparerName();
```

Это поле позволяет получать экземпляр класса `PersonComparerName` с применением `PersonComparerName.Default`, подобно показанному ранее классу `Comparer`. Метод `CompareTo()` в этом классе выглядит следующим образом:

```
public int Compare(object x, object y)
{
  if (x is Person && y is Person)
  {
    return Comparer.Default.Compare(
      ((Person)x).Name, ((Person)y).Name);
  }
  else
  {
    throw new ArgumentException(
       "One or both objects to compare are not Person objects.");
     // Один или оба сравниваемых объекта не являются объектами Person
  }
}
```

Вначале аргументы проверяются на предмет того, являются ли они объектами `Person`. Если нет, генерируется исключение. Если да, за счет применения стандартного объекта `Comparer` сравниваются значения строковых полей `Name` этих двух объектов `Person`.

Преобразования

До сих пор приведение использовалось только для преобразования одного типа в другой, но это не является единственным его применением. Точно так же, как тип `int` можно преобразовать в тип `long` или `double` неявным образом в виде части вычислений, классы

можно определять так, чтобы они могли быть преобразованы (либо явно, либо неявно) в другие классы. Для этого должны перегружаться операции преобразования, во многом подобно другим операциям, перегрузка которых демонстрировалась ранее в главе. Как это делается, будет показано в первой части раздела. Во второй части рассматривается еще одна полезная операция as, которую предпочтительно использовать для приведения ссылочных типов.

Перегрузка операций преобразования

Помимо перегрузки математических операций так, как было показано ранее в главе, можно определять явные и неявные операции преобразования между типами. Это необходимо для выполнения преобразования между типами, которые не связаны между собой, т.е. они, например, не имеют ни отношений наследования, ни разделяемых интерфейсов.

Для примера предположим, что требуется определить неявное преобразование между типами ConvClass1 и ConvClass2. Это значит, что можно будет записать так:

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = op1;
```

В качестве альтернативы можно определить явное преобразование:

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = (ConvClass2)op1;
```

Давайте рассмотрим следующий код:

```
public class ConvClass1
{
  public int val;
  public static implicit operator ConvClass2(ConvClass1 op1)
  {
    ConvClass2 returnVal = new ConvClass2();
    returnVal.val = op1.val;
    return returnVal;
  }
}

public class ConvClass2
{
  public double val;
  public static explicit operator ConvClass1(ConvClass2 op1)
  {
    ConvClass1 returnVal = new ConvClass1();
    checked {returnVal.val = (int)op1.val;};
    return returnVal;
  }
}
```

Здесь класс ConvClass1 содержит значение типа int, а ConvClass2 — значение типа double. Поскольку значения int могут быть преобразованы в значения double неявно, выполнение неявного преобразования между ConvClass1 и ConvClass2 возможно. Однако обратное неверно, поэтому операция преобразования между ConvClass2 и ConvClass1 должна быть определена как явная.

Это делается с использованием ключевых слов implicit и explicit, как было показано выше. Имея приведенные определения классов, можно записать следующий код, который будет работать нормально:

```
ConvClass1 op1 = new ConvClass1();
op1.val = 3;
ConvClass2 op2 = op1;
```

Для преобразования в обратном направлении, однако, должна применяться операция явного приведения:

```
ConvClass2 op1 = new ConvClass2();
op1.val = 3e15;
ConvClass1 op2 = (ConvClass1)op1;
```

Поскольку в операции явного преобразования было использовано слово `checked`, выполнение показанного выше кода приведет к генерации исключения, потому что значение свойства `val` объекта `op1` является слишком большим для того, чтобы уместиться в свойство `val` объекта `op2`.

Операция `as`

Операция `as` преобразует тип в указанный ссылочный тип с использованием следующего синтаксиса:

```
<операнд> as <тип>
```

Это возможно только при определенных условиях.

- Если <*операнд*> относится к типу <*тип*>.
- Если <*операнд*> может быть неявно преобразован в тип <*тип*>.
- Если <*операнд*> может быть упакован в тип <*тип*>.

Если преобразование <*операнд*> в <*тип*> невозможно, результатом выражения будет `null`.

Преобразование из базового класса в производный возможно с использованием явной операции преобразования, однако это работает не всегда. Возьмем классы `ClassA` и `ClassD` из предыдущего примера, где `ClassD` унаследован от `ClassA`:

```
class ClassA : IMyInterface
{
}

class ClassD : ClassA
{
}
```

В следующем коде операция `as` применяется для преобразования экземпляра `ClassA`, хранящегося в `obj1`, в `ClassD`:

```
ClassA obj1 = new ClassA();
ClassD obj2 = obj1 as ClassD;
```

В результате значением `obj2` окажется `null`.

За счет применения полиморфизма, однако, можно сделать так, чтобы экземпляры `ClassD` сохранялись в переменных типа `ClassA`. В следующем коде иллюстрируется этот подход, и операция `as` используется для преобразования в тип `ClassD` переменной типа `ClassA`, содержащей экземпляр типа `ClassD`:

```
ClassD obj1 = new ClassD();
ClassA obj2 = obj1;
ClassD obj3 = obj2 as ClassD;
```

На этот раз выполнение кода приведет к тому, что в `obj3` будет содержаться ссылка на тот же самый объект, а не `null`.

Такая функциональность делает операцию `as` очень полезной, поскольку, например, следующий код (в котором применяется простая операция приведения типов) будет приводить к генерации исключения:

```
ClassA obj1 = new ClassA();
ClassD obj2 = (ClassD)obj1;
```

Если в этом коде воспользоваться операцией as, исключение генерироваться не будет, а obj2 получит значение null. Поэтому код вроде показанного ниже (и в котором участвуют два разработанных ранее в главе класса — Animal и производный от него Cow) очень часто можно встретить в приложениях C#:

```
public void MilkCow(Animal myAnimal)
{
  Cow myCow = myAnimal as Cow;
  if (myCow != null)
  {
    myCow.Milk();
  }
  else
  {
    Console.WriteLine("{0} isn't a cow, and so can't be milked.",
       myAnimal.Name);
  }
}
```

Это намного проще, чем обеспечивать проверку исключений.

Резюме

В этой главе рассматривалось множество приемов, с помощью которых можно делать ООП-приложения гораздо более мощными и интересными. Хотя для их применения понадобится приложить определенные усилия, эти приемы помогают сделать классы гораздо более удобными для работы и, следовательно, упростить написание необходимого кода.

С каждым из рассмотренных здесь приемов связано несколько применений. С коллекциями наверняка придется иметь дело в той или иной форме практически в любом приложении, а создание строго типизированных коллекций может существенно облегчить жизнь, когда необходимо иметь дело с группой объектов одинакового типа. Также в главе рассказывалось о добавлении индексаторов и итераторов, предназначенных для доступа к объектам, находящимся внутри коллекций.

Сравнения и преобразования являются еще одним приемом, которые будет применяться постоянно. В этой главе было показано, как выполнять различные операции сравнения, а также демонстрировались базовые процессы упаковки и распаковки. Была описана перегрузка операций сравнения и преобразования, а также показано, как применять все эти приемы вместе для реализации сортировки списка.

Следующая глава посвящена обобщениям. Обобщения позволяют создавать классы, способные автоматически настраиваться на работу с динамически выбранными типами. Они особенно полезны в коллекциях, и вы увидите, что обобщенные коллекции могут существенно упростить большую часть кода, приведенного в настоящей главе.

Упражнения

1. Создайте класс коллекции по имени People для хранения экземпляров приведенного ниже класса Person. Элементы в этой коллекции должны быть доступны через строковый индексатор, в роли которого должно выступать имя человека, идентичное хранящемуся в свойстве Person.Name:

```
public class Person
{
  private string name;
  private int age;
```

```
    public string Name
    {
      get
      {
        return name;
      }
      set
      {
        name = value;
      }
    }
    public int Age
    {
      get
    {
      return age;
    }
    set
    {
      age = value;
    }
  }
}
```

2. Расширьте созданный в предыдущем упражнении класс `Person` так, чтобы в нем перегружались операции >, <, >= и <= и выполнялось сравнение свойства `Age` со свойством `Age` других экземпляров `Person`.
3. Добавьте в класс `People` метод `GetOldest`, предусматривающий возврат массива объектов `Person` с наибольшим значением в свойстве `Age` (их может быть один и более, поскольку многие элементы могут иметь в этом свойстве одинаковые значения), с использованием перегруженных операций из упражнения 2.
4. Реализуйте в классе `People` интерфейс `ICloneable` для обеспечения возможности глубокого копирования.
5. Добавьте в класс `People` итератор, позволяющий получать значения `Age` всех членов в цикле `foreach` показанным ниже образом:

```
foreach (int age in myPeople.Ages)
{
  // Отображение значений возраста.
}
```

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Определение коллекций	Коллекции — это классы, которые содержат экземпляры других классов. Класс коллекции можно определить, унаследовав его от `CollectionBase` или самостоятельно реализовав интерфейсы коллекций, такие как `IEnumerable`, `ICollection` и `IList`. Обычно для коллекции определяется индексатор, который позволяет использовать для доступа к элементам синтаксис *`коллекция`*`.[`*`индекс`*`]`.
Словари	Можно также определять коллекции с ключами, или словари, в которых каждый элемент имеет ассоциированный с ним ключ. В этом случае ключ может использоваться для идентификации элемента вместо индекса. Словарь можно определить за счет реализации интерфейса `IDictionary` или наследования класса от `DictionaryBase`.
Итераторы	Для управления тем, как код цикла извлекает значения при каждом проходе, можно реализовать итератор. Для итерации по классам понадобится реализовать метод `GetEnumerator()` с типом возврата `IEnumerator`. Для итерации по членам класса, таким как методы, в качестве типа возврата должен использоваться `IEnumerable`. В блоках кода итераторов значения возвращаются с помощью ключевого слова `yield`.
Сравнения типов	Метод `GetType()` позволяет получить тип объекта, а операция `typeof` — тип класса. Значения этих типов можно сравнивать. С помощью операции `is` можно определить, совместим ли объект с определенным типом класса.
Сравнения значений	Чтобы экземпляры классов могли сравниваться с использованием стандартных операций C#, необходимо перегрузить эти операции в определении класса. Для других типов сравнения значений можно применять классы, которые реализуют интерфейс `IComparable` или `IComparer`. В частности, упомянутые интерфейсы полезны для сортировки коллекций.
Операция `as`	Операцию `as` можно использовать для преобразования значения в ссылочный тип. Если преобразование невозможно, операция `as` возвращает `null`.

12

Обобщения

В ЭТОЙ ГЛАВЕ...

- Что собой представляют обобщения
- Использование некоторых обобщенных классов, поставляемых в составе .NET Framework
- Определение собственных обобщений
- Применение вариантности в обобщениях

Одной из (довольно немногочисленных) жалоб по поводу первой версии языка C# было отсутствие поддержки *обобщений* (generics). Обобщения в языке C++ (называемые *шаблонами*) давно считаются замечательным приемом, поскольку позволяют единственному определению типа превращаться во множество специализированных типов во время компиляции и тем самым экономить огромное количество времени и усилий. По какой-то причине обобщения не вошли в первую версию языка C#, и он из-за этого пострадал. Возможно, так случилось потому, что обобщения часто находят довольно сложными для изучения, или потому, что их сочли не столь уж необходимыми. К счастью, начиная с версии C# 2.0, они стали доступны. Даже еще лучше то, что их не так трудно использовать, хотя для этого требуется смотреть на вещи несколько по-иному.

Глава начинается с рассмотрения понятия обобщений. Знание лежащих в их основе понятий критически важно для эффективного их использования.

Кроме того, будут демонстрироваться в действии некоторые из обобщенных типов, входящие в состав .NET Framework. Это позволит ознакомиться с их функциональными возможностями и новым синтаксисом, который должен использоваться в коде. Также будет показано, как определять собственные обобщенные типы, в том числе обобщенные классы, интерфейсы, методы и делегаты, и описаны дополнительные приемы для их настройки, к числу каковых относятся ключевое слово `default` и возможность ограничения типов.

И, наконец, в главе рассматриваются понятия ковариантности и контравариантности, которые являются новинкой в C# 4 и обеспечивают большую гибкость при использовании обобщенных классов.

Что собой представляют обобщения

Лучше всего понять, что собой представляют обобщения и почему они настолько полезны, можно, вспомнив классы коллекций, которые демонстрировались в предыдущей главе. Как было показано, базовые коллекции могут содержаться в классах, подобных `ArrayList`, однако они не типизированы и потому требуют приведения элементов `object` к нужным типам. Из-за того, что в `ArrayList` могут храниться любые объекты, унаследованные от `System.Object` (т.е. практически какие угодно), необходимо соблюдать осторожность. Предположение о том, что в коллекции содержатся объекты только определенных типов, может заканчиваться генерацией исключений и нарушением логики кода. В предыдущей главе демонстрировались приемы, с помощью которых можно обходить подобные проблемы, включая код, используемый для проверки типа объекта.

Однако также выяснилось, что гораздо более эффективным решением является применение с самого начала строго типизированной коллекции. За счет наследования от класса `CollectionBase` и предоставления собственных методов для добавления, удаления и получения доступа к членам коллекции можно ограничивать члены коллекции только теми, которые наследуются от определенного базового типа или поддерживают определенный интерфейс. Именно здесь и начинаются проблемы. При каждом создании нового класса, который должен содержаться в коллекции, необходимо предпринимать одно из описанных ниже действий.

- Использовать созданный класс коллекции, который может содержать элементы нового типа.
- Создать новый класс коллекции, который может хранить элементы нового типа, реализовав все требуемые методы.

Обычно вместе с новым типом требуется и какая-то дополнительная функциональность, поэтому чаще всего будет создаваться новый класс коллекции. По этой причине подготовка классов коллекций может отнимать немало времени.

Обобщенные классы, напротив, значительно упрощают дело. Обобщенный класс создается вокруг типа или типов, предоставляемых во время создания экземпляра, и тем самым обеспечивает строгую типизацию практически безо всяких усилий. В контексте коллекций создание "коллекции объектов типа `T`" выглядит так же просто и требует написания всего одной строки кода. То есть вместо такого кода:

```
CollectionClass items = new CollectionClass();
items.Add(new ItemClass());
```

может использоваться следующий код:

```
CollectionClass<ItemClass> items = new CollectionClass<ItemClass>();
items.Add(new ItemClass());
```

Угловые скобки в синтаксисе представляют собой способ передачи обобщенным типам параметров типа. В показанном выше коде `CollectionClass<ItemClass>` следует воспринимать как коллекция `CollectionClass` из элементов `ItemClass`. Этот синтаксис будет более подробно рассматриваться позже в этой главе.

Обобщения охватывают не только коллекции, но для них они подходят больше всего, как будет показано позже в главе при рассмотрении пространства имен `System.Collections.Generic`. За счет создания обобщенного класса можно генерировать методы со строго типизированной сигнатурой, даже с учетом того, что типом может быть тип значения или ссылочный тип, и иметь дело с особыми случаями. Можно делать допустимым только подмножество типов, ограничивая типы, применяемые для создания экземпляра обобщенного класса, только теми, которые поддерживают конкретный интерфейс или наследуются от конкретного типа. Более того, можно создавать не только обобщенные классы, но также обобщенные интерфейсы, обобщенные методы (которые можно определять в не обобщенных классах) и даже обобщенные делегаты. Все это позволяет делать код гораздо более гибким и экономит многие часы при его разработке.

К этому моменту наверняка возник вопрос о том, каким образом все это становится возможным. Обычно созданный класс компилируется в тип, который затем можно использовать в своем коде. Из-за этого может показаться, что созданный обобщенный класс должен компилироваться во множество типов, чтобы затем можно было создавать его экземпляры. К счастью, все происходит не так, а с учетом того, что количество классов, которые можно создавать в .NET, практически бесконечно, очень хорошо, что это не так. Исполняющая среда .NET "за кулисами" позволяет обобщенным классам генерироваться динамически, только когда в них возникает необходимость. Никакого обобщенного класса `A` для класса `B` не будет существовать до тех пор, пока он не будет запрошено создание его экземпляра.

Тем кто, знаком с языком C++ или интересуется им, следует обратить внимание на одно отличие между шаблонами в C++ и обобщенными классами в C#. В C++ компилятор определяет, где в коде был использован шаблон специфического типа, например, шаблон `A` для класса `B`, и компилирует необходимый для создания такого типа код. В C# весь этот процесс происходит во время выполнения.

Вывод из всего сказанного выше: обобщения позволяют создавать гибкие типы, способные обрабатывать объекты одного и более конкретных типов, причем эти типы определяются лишь при создании экземпляра обобщения или его использовании каким-то другим образом. Давайте посмотрим на обобщения в действии.

Использование обобщений

Прежде чем приступить к изучению того, как создавать собственные обобщения, не помешает посмотреть, какие типы подобного рода предлагаются в .NET Framework. К ним относятся типы из пространства имен `System.Collections.Generic`, которое уже не-

сколько раз встречалось в коде, потому что оно по умолчанию включается в состав консольных приложений. До сих пор типы из этого пространства имен не использовались. В настоящем разделе будет показано, как применять эти типы для создания строго типизированных коллекций и усовершенствования функциональности существующих.

Однако сначала рассмотрим *нулевые типы* (nullable types) — еще один простой обобщенный тип, позволяющий обходить упомянутую ранее проблему с типами значения.

Нулевые типы

В предыдущих главах было показано, что одно из отличий между типами значения (к числу которых относится большинство базовых типов, таких как `int` и `double`, а также структуры) и ссылочными типами (вроде `string` и любого класса) состоит в том, что типы значения должны обязательно содержать какое-то значение. Они могут существовать в неприсвоенном состоянии в промежутке между их объявлением и присваиванием значения, но пользоваться типом значения в этом состоянии нельзя. В отличие от них, ссылочные типы могут быть нулевыми (т.е. `null`).

В ряде ситуаций, которые возникают гораздо чаще, чем может показаться (особенно во время работы с базами данных), удобно иметь тип значения, способный принимать `null`. Обобщения предлагают способ для получения такого типа, и заключается он в использовании типа `System.Nullable<T>`, как показано в следующем примере:

```
System.Nullable<int> nullableInt;
```

В этой строке кода объявляется переменная по имени `nullableInt`, которая может принимать все значения типа `int` плюс `null`. Это позволяет использовать код, подобный показанному ниже:

```
nullableInt = null;
```

Если бы `nullableInt` не представляла собой переменную типа `int`, скомпилировать бы предыдущий код не удалось.

Предыдущая операция присваивания эквивалента такой строке кода:

```
nullableInt = new System.Nullable <int> ();
```

Подобно любой другой, эту переменную нельзя использовать без предварительной инициализации, пусть даже значением `null` (с помощью показанного ранее синтаксиса).

Точно так же, как и ссылочные типы, нулевые типы можно проверять на предмет равенства `null`:

```
if (nullableInt == null)
{
  ...
}
```

В качестве альтернативы можно использовать свойство `HasValue`:

```
if (nullableInt.HasValue)
{
  ...
}
```

Для ссылочных типов такой код работать не будет, даже для тех, у которых имеется собственное свойство `HasValue`, поскольку наличие ссылочного типа со значением `null` означает, что никакого объекта, через который можно было бы получить доступ к данному свойству, не существует, и потому будет генерироваться исключение.

Значение нулевого типа можно просматривать с использованием свойства `Value`. Если свойство `HasValue` равно `true`, значением свойства `Value` является не `null`, но если

HasValue равно false — т.е. переменной было присвоено значение null — доступ к Value повлечет за собой генерацию исключения System.InvalidOperationException.

Следует отметить, что нулевые типы настолько полезны, что привели даже к изменению синтаксиса C#. Вместо показанного выше синтаксиса для объявления переменной нулевого типа теперь можно использовать и такой синтаксис:

```
int? nullableInt;
```

Здесь int? представляет собой просто сокращенный вариант синтаксиса System.Nullable <int>, но гораздо более удобно для восприятия, и потому в последующих разделах будет применяться именно этот синтаксис.

Операции и нулевые типы

В случае простых типов, таких как int, для работы с их значениями можно применять операции вроде +, - и т.д. Для эквивалентных им нулевых типов дела обстоят точно так же: значения, содержащиеся в нулевых типах, неявно преобразуются в требуемый тип, после чего подвергаются надлежащим операциям. То же самое касается структур с предоставляемыми для них операциями:

```
int? op1 = 5;
int? result = op1 * 2;
```

Обратите внимание, что здесь переменная result тоже относится к типу int?. Поэтому следующий код компилироваться не будет:

```
int? op1 = 5;
int result = op1 * 2;
```

Чтобы заставить этот код работать, необходимо добавить явное преобразование или получать доступ к значению через свойство Value, т.е. использовать следующий код:

```
int? op1 = 5;
int result = (int)op1 * 2;
```

либо такой код:

```
int? op1 = 5;
int result = op1.Value * 2;
```

Этот код работает нормально, когда op1 имеет какое-то значение: в случае если op1 равно null, генерируется исключение System.InvalidOperationException.

Возникает вполне естественный вопрос: а что произойдет, если в процессе вычисления операции обнаружится, что одно или оба значения равны null, как op1 в показанном ниже коде?

```
int? op1 = null;
int? op2 = 5;
int? result = op1 * op2;
```

Ответ выглядит так: для всех простых нулевых типов кроме bool? вычисление операции будет завершаться возвратом значения null, которое свидетельствует о том, что произвести вычисление не удалось. В случае структур для обработки подобных ситуаций можно определить собственные операции (это будет показано далее в главе), а для типа bool? — использовать предопределенные операции & и |, которые возвращают отличные от null значения и перечислены в табл. 12.1.

Результаты, приведенные в этой таблице, вполне понятны с логической точки зрения: если есть достаточно информации, чтобы вывести ответ вычисления без знания значения одного из операндов, факт наличия в нем null становится неважным.

Таблица 12.1. Выполнение операций & и | для операндов типа bool?

op1	op2	op1 & op2	op1 | op2
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Операция ??

Для дальнейшего сокращения объема кода, необходимого для работы с нулевыми типами, и упрощения кода, который имеет дело с переменными, способными принимать значения null, можно использовать операцию ??. Эта бинарная операция, которая также называется *операцией объединения с* null, позволяет предоставлять альтернативное значение для использования в выражениях, которые могут при вычислении возвращать null. Если значением первого операнда оказывается не null, она возвращает значение этого операнда, а если null — значение второго операнда. Два следующих выражения функционально эквивалентны:

```
op1 ?? op2
op1 == null ? op2 : op1
```

В этом коде op1 может быть любым нулевым выражением, в том числе ссылочного и, что более важно, нулевого типа. Это означает, что операцию ?? можно использовать для предоставления значений, которые должны использоваться по умолчанию в случае, если переменная нулевого типа оказывается равной null:

```
int? op1 = null;
int result = op1 * 2 ?? 5;
```

Из-за того, что op1 в этом примере присваивается null, выражение op1 * 2 тоже дает null. Однако операция ?? обнаруживает это и присваивает result значение 5. Очень важно обратить здесь внимание на отсутствие какого-либо явного преобразования для помещения результата в целочисленную переменную result. Операция ?? выполняет необходимое преобразование. В качестве альтернативы результат вычисления ?? можно поместить в int?:

```
int? result = op1 * 2 ?? 5;
```

Такое поведение делает операцию ?? универсальной для применения в работе с нулевыми переменными и удобной для предоставления значений по умолчанию, без необходимости писать блок кода в структуре if либо иметь дело с часто вызывающей путаницу тернарной операцией.

Следующее практическое занятие посвящено экспериментам с нулевым типом Vector.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Нулевые типы

1. Создайте новый проект типа консольного приложения по имени Ch12Ex01 и сохраните его в каталоге C:\BegVCSharp\Chapter12.

2. Добавьте в него новый класс `Vector` в файле `Vector.cs`.
3. Измените код в файле `Vector.cs` следующим образом:

```
public class Vector
{
  public double? R = null;
  public double? Theta = null;
  public double? ThetaRadians
  {
    get
    {
      // Преобразование градусов в радианы.
      return (Theta * Math.PI / 180.0);
    }
  }
  public Vector(double? r, double? theta)
  {
  // Нормализация.
  if (r < 0)
  {
    r = -r;
    theta += 180;
  }
  theta = theta % 360;

  // Установка полей.
  R = r;
  Theta = theta;
}
public static Vector operator +(Vector op1, Vector op2)
{
  try
  {
     // Получение координат (x, y) для нового вектора.
     double newX = op1.R.Value * Math.Sin(op1.ThetaRadians.Value)
       + op2.R.Value * Math.Sin(op2.ThetaRadians.Value);
     double newY = op1.R.Value * Math.Cos(op1.ThetaRadians.Value)
        + op2.R.Value * Math.Cos(op2.ThetaRadians.Value);

     // Преобразование в (r, theta).
     double newR = Math.Sqrt(newX * newX + newY * newY);
     double newTheta = Math.Atan2(newX, newY) * 180.0 / Math.PI;

     // Возврат результата.
     return new Vector(newR, newTheta);
   }
   catch
   {
     // Возврат "нулевого" вектора.
     return new Vector(null, null);
   }
  }
  public static Vector operator -(Vector op1)
  {
    return new Vector(-op1.R, op1.Theta);
  }
  public static Vector operator -(Vector op1, Vector op2)
  {
    return op1 + (-op2);
  }
```

```
  public override string ToString()
  {
    // Получение строкового представления координат.
    string rString = R.HasValue ? R.ToString() : "null";
    string thetaString = Theta.HasValue ? Theta.ToString() : "null";

    // Возврат строки (r, theta).
    return string.Format("({0}, {1})", rString, thetaString);
  }
}
```

Фрагмент кода ***Ch12Ex01\Vector.cs***

4. Измените код в файле `Program.cs` так, как показано ниже:

```
class Program
{
  static void Main(string[] args)
  {
    Vector v1 = GetVector("vector1");
    Vector v2 = GetVector("vector1");
    Console.WriteLine("{0} + {1} = {2}", v1, v2, v1 + v2);
    Console.WriteLine("{0} - {1} = {2}", v1, v2, v1 - v2);
    Console.ReadKey();
  }
  static Vector GetVector(string name)
  {
    Console.WriteLine("Input {0} magnitude:", name);
                    // Ввод абсолютной величины вектора
    double? r = GetNullableDouble();
    Console.WriteLine("Input {0} angle (in degrees):", name);
                    // Ввод угла вектора (в градусах)
    double? theta = GetNullableDouble();
    return new Vector(r, theta);
  }
  static double? GetNullableDouble()
  {
    double? result;
    string userInput = Console.ReadLine();
    try
    {
      result = double.Parse(userInput);
    }
    catch
    {
      result = null;
    }
    return result;
  }
}
```

Фрагмент кода ***Ch12Ex01\Program.cs***

5. Запустите приложение и введите значения абсолютной величины и угла для двух векторов. На рис. 12.1 показан пример результата, который должен получиться.
6. Запустите приложение снова, но на этот раз пропустите ввод какого-то из четырех требуемых значений. На рис. 12.2 показан пример результата, который получится в таком случае.

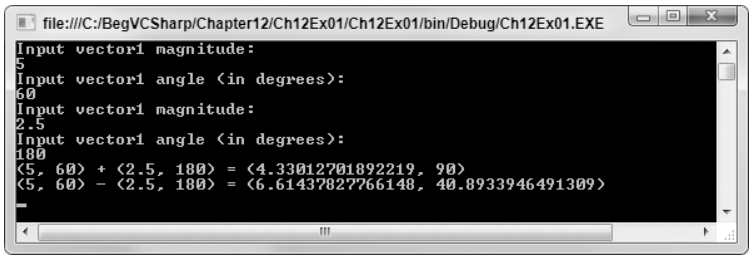


Рис. 12.1. *Приложение* `Ch12Ex01` *в действии*

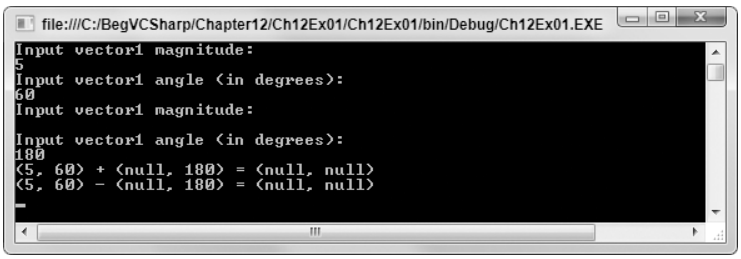


Рис. 12.2. *Пропуск ввода одного из значений в приложении* `Ch12Ex01`

Описание работы

В этом примере был создан класс по имени `Vector`, представляющий вектор с полярными координатами (абсолютной величиной и углом), как показано на рис. 12.3.

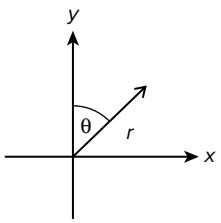


Рис. 12.3. *Вектор с полярными координатами*

Координаты *r* и θ представлены в коде общедоступными полями `R` и `Theta`, причем значение поля `Theta` выражается в градусах. Для получения значения `Theta` в радианах предоставляется `ThetaRad`; это необходимо потому, что в статических методах класса `Math` используются радианы. И `R`, и `Theta` являются значениями типа `double?`, а это значит, что они могут быть равны `null`:

```
public class Vector
{
  public double? R = null;
  public double? Theta = null;

  public double? ThetaRadians
  {
    get
    {
      // Преобразование градусов в радианы.
      return (Theta * Math.PI / 180.0);
    }
  }
```

Фрагмент кода ***Ch12Ex01\Vector.cs***

В конструкторе класса `Vector` сначала производится нормализация первоначальных значений `R` и `Theta`, а затем установка соответствующих общедоступных полей:

```
public Vector(double? r, double? theta)
{
  // Нормализация.
```

```
    if (r < 0)
    {
      r = -r;
      theta += 180;
    }
    theta = theta % 360;
    // Установка полей.
    R = r;
    Theta = theta;
  }
```

Главная функциональность класса `Vector` состоит в выполнении сложения и вычитания векторов с применением перегруженных операций, что требует наличия базовых знаний тригонометрии. Наиболее важным в этом коде является то, что в случае выдачи исключения при получении значения свойства `Value` поля `R` или `ThetaRadians` — т.е. когда какое-то из них равно `null` — он предусматривает возврат "нулевого" вектора:

```
public static Vector operator +(Vector op1, Vector op2)
{
  try
  {
    // Получение координат (x, y) для нового вектора.
    ...
  }
  catch
  {
    // Возврат "нулевого" вектора.
    return new Vector(null, null);
  }
}
```

Если какая-то из образующих вектор координат равна `null`, вектор получается недействительным, что здесь выражается в возврате экземпляра класса `Vector` со значениями `null` в обоих полях (`R` и `Theta`). В остальной части кода в классе `Vector` переопределяются другие операции для расширения функции сложения так, чтобы она могла выполнять и вычитание, а также метод `ToString()` для получения строкового представления объекта `Vector`.

В коде `Program.cs` производится тестирование класса `Vector` за счет предоставления пользователю возможности инициализировать два вектора и затем выполнить их сложение и вычитание. Если какое-то значение пользователем опущено, оно интерпретируется как `null`, и тогда применяются упоминавшиеся ранее правила.

Пространство имен `System.Collections.Generics`

Практически в каждом из демонстрировавшихся ранее приложений встречались следующие пространства имен:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

В пространстве имен `System` содержится большая часть базовых типов, которые применяются в приложениях .NET. Пространство имен `System.Text` включает типы, имеющие отношение к обработке и кодированию строк. О пространстве имен `System.Linq` речь пойдет позже в настоящей книге, начиная с главы 23. А какие типы доступны в пространстве имен `System.Collections.Generic` и почему оно включается в состав консольных приложений по умолчанию?

Ответ на этот вопрос можно сформулировать так: это пространство имен содержит обобщенные типы для работы с коллекциями, и вероятность его использования настолько высока, что оно заранее указано в операторе `using`, который позволяет применять его без явного указания имени.

Как обещалось ранее в главе, сейчас мы посмотрим, какие типы содержатся в данном пространстве имен и как они могут упростить разработку. Эти типы позволяют создавать строго типизированные классы коллекций практически безо всяких усилий. В табл. 12.2 приведено краткое описание двух типов из пространства имен `System.Collections.Generics`, о которых пойдет речь в настоящем разделе. Другие типы из этого пространства имен рассматриваются позже в этой главе.

Таблица 12.2. Два типа из пространства имен `System.Collections.Generics`

Тип	Описание
`List<T>`	Коллекция объектов типа `T`
`Dictionary<K, V>`	Коллекция элементов типа `V`, ассоциируемых с ключами типа `K`

В данном разделе также будут показаны различные интерфейсы и делегаты, которые могут использоваться с этими классами.

Тип `List<T>`

Иногда вместо наследования класса от `CollectionBase` и реализации необходимых методов, как это делалось в предыдущей главе, быстрее и проще воспользоваться обобщенным типом коллекции `List<T>`. Дополнительное преимущество такого подхода состоит в том, что при его применении многие методы, которые обычно должны быть реализованы самостоятельно, такие как `Add()`, реализуются автоматически.

Создание коллекции объектов типа `T` осуществляется следующим образом:

```
List<T> myCollection = new List<T>();
```

И это все! Ни определять какие-либо классы, ни реализовать методы, ни делать еще что-нибудь еще не требуется. Можно также установить начальный список элементов в коллекции, передав конструктору объект `List<T>`. Объект, экземпляр которого создается с помощью такого синтаксиса, поддерживает методы и свойства, перечисленные в табл. 12.3 (где подразумевается, что типом, предоставляемым обобщению `List<T>`, является `T`).

Таблица 12.3. Методы и свойства `List<T>`

Член	Описание
`int Count`	Свойство, предоставляющее количество элементов в коллекции
`void Add(T item)`	Добавляет в коллекцию один элемент
`void AddRange(IEnumerable<T>)`	Добавляет в коллекцию несколько элементов
`IList<T> AsReadOnly()`	Возвращает доступный только для чтения интерфейс для коллекции
`int Capacity`	Получает или устанавливает количество элементов, которое может содержаться в коллекции
`void Clear()`	Удаляет все элементы из коллекции
`bool Contains(T item)`	Определяет, содержится ли данный элемент в коллекции

Окончание табл. 12.3

Член	Описание
`void CopyTo(T[] array, int index)`	Копирует элементы из коллекции в массив `array`, начиная с индекса `index` в массиве
`IEnumerator<T> GetEnumerator()`	Возвращает экземпляр `IEnumerator<T>` для выполнения итерации по коллекции. Обратите внимание, что возвращаемый интерфейс является строго типизированным и относится к типу `T`, поэтому в циклах `foreach` никакого приведения не требуется
`int IndexOf(T item)`	Возвращает индекс элемента или `-1`, если элемент не содержится в коллекции
`void Insert(int index, T item)`	Вставляет в коллекцию элемент по указанной позиции `index`
`bool Remove(T item)`	Удаляет из коллекции первое вхождение элемента и возвращает `true`. Если элемент не содержится в коллекции, тогда возвращает `false`
`void RemoveAt(int index)`	Удаляет из коллекции элемент, который находится в позиции `index`

Класс `List<T>` имеет также свойство `Item`, которое позволяет получать доступ к элементам коллекции подобным массивам образом:

```
T itemAtIndex2 = myCollectionOfT[2];
```

Этот класс также поддерживает несколько других методов, но рассмотренных выше вполне достаточно для работы с ним. В следующем практическом занятии будет показано, как `List<T>` применять на практике.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Применение типа `List<T>`

1. Создайте новое консольное приложение по имени `Ch12Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter12`.
2. Щелкните правой кнопкой мыши на имени этого проекта в окне Solution Explorer и выберите в контекстном меню пункт Add⇨Existing Item (Добавить⇨Существующий элемент).
3. Выберите из каталога `C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01` файлы `Animal.cs`, `Cow.cs` и `Chicken.cs` и щелкните на кнопке Add (Добавить).
4. Измените объявление пространства имен в каждом из трех добавленных файлов следующим образом:

```
namespace Ch12Ex02
```

*Фрагмент кода **Ch12Ex02\Animal.cs, Ch12Ex02\Cow.cs и Ch12Ex02\Chicken.cs***

5. Модифицируйте код в файле `Program.cs`, как показано ниже:

```
static void Main(string[] args)
{
  List<Animal> animalCollection = new List<Animal>();
  animalCollection.Add(new Cow("Jack"));
  animalCollection.Add(new Chicken("Vera"));
```

```
    foreach (Animal myAnimal in animalCollection)
    {
      myAnimal.Feed();
    }
    Console.ReadKey();
  }
```

Фрагмент кода ***Ch12Ex02\Program.cs***

6. Запустите приложение. Результат должен получиться точно таким же, как приложении Ch11Ex02, которое рассматривалось в предыдущей главе.

Описание работы

Этот пример отличается от примера Ch11Ex02 только двумя деталями. Первая состоит в том, что строка кода

```
Animals animalCollection = new Animals();
```

была заменена следующей строкой:

```
List<Animal> animalCollection = new List<Animal>();
```

Второе и более важное отличие связано с отсутствием в проекте класса коллекции Animals. Благодаря использованию обобщенного класса коллекции, вся трудная работа, которая выполнялась в предыдущем примере для создания этого класса, здесь была сведена к единственной строке кода.

В качестве альтернативы добиться того же результата можно, оставив код в Program.cs неизменным и воспользовавшись следующим определением для класса Animals:

```
public class Animals : List <Animal>
{
}
```

Преимущество такого подхода в том, что он несколько упрощает код в файле Program.cs и при необходимости позволяет добавлять в класс Animals дополнительные члены.

К этому моменту может возникнуть хороший вопрос о том, когда вообще требуется наследовать классы от CollectionBase. На самом деле, ситуаций, в которых это действительно необходимо, существует не очень много. Разумеется, знать внутреннюю работу класса CollectionBase полезно, потому что List<T> работает во многом аналогично, но по большей части класс CollectionBase служит для обеспечения обратной совместимости. Единственными обстоятельствами для применения класса CollectionBase является необходимость иметь больший контроль над предоставляемыми членами класса. Например, если требуется класс коллекции с модификатором доступа internal в методе Add(), то использование CollectionBase может быть наилучшим вариантом.

Конструктору List<T> можно также передавать значение первоначальной емкости списка (в виде int) или начальный список элементов с использованием интерфейса IEnumerable<T>. Класс List<T> входит в набор классов, которые поддерживают этот интерфейс.

Сортировка и поиск в обобщенных списках

Сортировка обобщенного списка выполняется во многом так же, как и сортировка любого другого списка. В предыдущей главе было показано, как использовать интерфейсы IComparer и IComparable для сравнения двух объектов и за счет этого — сортировки списка объектов такого типа. Единственное отличие здесь связано с тем, что применять нужно обобщенные интерфейсы IComparer<T> и IComparable<T>, которые предоставляют немного отличающиеся методы для каждого типа. Все отличия между этими методами кратко описаны в табл. 12.4.

Таблица 12.4. Отличия между обобщенными и не обобщенными методами

Обобщенный метод	Не обобщенный метод	Отличие
`int IComparable<T>. CompareTo(T otherObj)`	`int IComparable. CompareTo(object otherObj)`	В обобщенных версиях является строго типизированным
`bool IComparable<T>. Equals(T otherObj)`	—	Не существует в не обобщенном интерфейсе; вместо него может использоваться унаследованный метод `object.Equals()`
`int IComparer<T>. Compare(T objectA, T objectB)`	`int IComparer.Compare (object objectA, object object B)`	В обобщенных версиях является строго типизированным
`bool IComparer<T>. Equals(T objectA, T objectB)`	—	Не существует в не обобщенном интерфейсе; вместо него может использоваться унаследованный метод `object.Equals()`
`object.Equals() instead int IComparer<T>. GetHashCode(T objectA)`	—	Не существует в не обобщенном интерфейсе; вместо него может использоваться унаследованный метод `object.GetHashCode()`

Для выполнения сортировки `List<T>` можно предоставить интерфейс `IComparable<T>` прямо внутри сортируемого типа либо интерфейс `IComparer<T>`. Кроме того, в качестве метода для сортировки можно указывать *обобщенный делегат*. Последний подход является гораздо более интересным, поскольку реализация упомянутых выше интерфейсов, по сути, ничем не отличается от реализации их не обобщенных аналогов.

В принципе для проведения сортировки списка необходимо иметь только метод, сравнивающий два объекта типа `T`, а для выполнения поиска — метод, проверяющий объект типа `T` на предмет соответствия конкретным критериям. Помогают в определении таких методов следующие два обобщенных типа-делегата.

- `Comparison<T>`. Тип делегата для метода, используемого для сортировки, который имеет следующий возвращаемый тип и параметры:

  ```
  int метод (T objectA, T objectB)
  ```

- `Predicate<T>`. Тип делегата для метода, используемого для поиска, который имеет следующий возвращаемый тип и параметр:

  ```
  bool метод (T targetObject)
  ```

Таких методов можно определить произвольное количество, после чего использовать их для "подключения" к методам сортировки и поиска `List<T>`. В следующем практическом занятии все это демонстрируется на практике.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Выполнение сортировки и поиска в `List<T>`

1. Создайте новое консольное приложение по имени `Ch12Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter12`.
2. Щелкните правой кнопкой мыши на имени этого проекта в окне Solution Explorer и выберите в контекстном меню пункт Add⇨Existing Item (Добавить⇨Существующий элемент).

3. Выберите из каталога `C:\BegVCSharp\Chapter12\Ch12Ex01\Ch12Ex01` файл `Vector.cs` и щелкните на кнопке Add (Добавить).
4. Измените объявление пространства имен в добавленном файле следующим образом:
   `namespace Ch12Ex03`
5. Добавьте новый класс по имени `Vectors`.
6. Измените код в файле `Vectors.cs` так, как показано ниже:

```
public class Vectors : List<Vector>
{
  public Vectors()
  {
  }
  public Vectors(IEnumerable<Vector> initialItems)
  {
    foreach (Vector vector in initialItems)
    {
      Add(vector);
    }
  }
  public string Sum()
  {
    StringBuilder sb = new StringBuilder();
    Vector currentPoint = new Vector(0.0, 0.0);
    sb.Append("origin");
    foreach (Vector vector in this)
    {
      sb.AppendFormat(" + {0}", vector);
      currentPoint += vector;
    }
    sb.AppendFormat(" = {0}", currentPoint);
    return sb.ToString();
  }
}
```

Фрагмент кода ***Ch12Ex03\Vector.cs***

7. Добавьте новый класс по имени `VectorDelegates`.
8. Модифицируйте код в файле `VectorDelegates.cs` следующим образом:

```
public static class VectorDelegates
{
  public static int Compare(Vector x, Vector y)
  {
    if (x.R > y.R)
    {
      return 1;
    }
    else if (x.R < y.R)
    {
      return -1;
    }
    return 0;
  }
  public static bool TopRightQuadrant(Vector target)
  {
    if (target.Theta >= 0.0 && target.Theta <= 90.0)
    {
      return true;
    }
```

```
      else
      {
        return false;
      }
    }
  }
```

Фрагмент кода ***Ch12Ex03\VectorDelegates.cs***

9. Измените код в файле `Program.cs` так, как показано ниже:

```
static void Main(string[] args)
{
  Vectors route = new Vectors();
  route.Add(new Vector(2.0, 90.0));
  route.Add(new Vector(1.0, 180.0));
  route.Add(new Vector(0.5, 45.0));
  route.Add(new Vector(2.5, 315.0));

  Console.WriteLine(route.Sum());

  Comparison<Vector> sorter = new Comparison<Vector>(VectorDelegates.Compare);
  route.Sort(sorter);
  Console.WriteLine(route.Sum());

  Predicate<Vector> searcher =
   new Predicate<Vector>(VectorDelegates.TopRightQuadrant);
  Vectors topRightQuadrantRoute = new Vectors(route.FindAll(searcher));
  Console.WriteLine(topRightQuadrantRoute.Sum());
  Console.ReadKey();
}
```

Фрагмент кода ***Ch12Ex03\Program.cs***

10. Запустите приложение. На рис. 12.4 показан результат, который должен получиться.

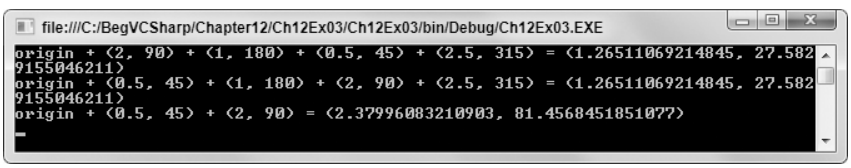


Рис. 12.4. *Приложение* `Ch12Ex03` *в действии*

Описание работы

В этом примере был создан класс коллекции по имени `Vectors` для класса `Vector` из примера `Ch12Ex01`. Можно было бы использовать просто переменную типа `List<Vector>`, но поскольку требуется дополнительная функциональность, строится новый класс `Vectors`, унаследованный от `List<Vector>`, и к нему добавляются новые члены.

Одним из таких членов является метод `Sum()`, возвращающий строку, в которой по очереди перечисляются все векторы, а также результат сложения всех этих векторов (с помощью перегруженной операции + из первоначального класса `Vector`). Поскольку каждый вектор может трактоваться как направление и расстояние, по сути, получается маршрут с конечной точкой.

```
public string Sum()
{
  StringBuilder sb = new StringBuilder();
  Vector currentPoint = new Vector(0.0, 0.0);
  sb.Append("origin");
```

```
    foreach (Vector vector in this)
    {
      sb.AppendFormat(" + {0}", vector);
      currentPoint += vector;
    }
    sb.AppendFormat(" = {0}", currentPoint);
    return sb.ToString();
  }
```

Фрагмент кода ***Ch12Ex03\Vector.cs***

В этом методе для формирования строки ответа применяется удобный класс `StringBuilder` из пространства имен `System.Text`. Члены этого класса, вроде используемых здесь `Append()` и `AppendFormat()`, позволяют легко собирать строку и получать более высокую производительность, чем при конкатенации отдельных строк. Для получения результирующей строки используется метод `ToString()` этого класса.

Кроме того, создаются еще два новых метода для применения в качестве делегатов — в виде статических членов `VectorDelegates`. Метод `Compare()` предназначен для сравнения (сортировки), а `TopRightQuadrant()` — для поиска. Более подробно они рассматриваются во время анализа кода `Program.cs`.

Код в `Main()` начинается с инициализации коллекции `Vectors`, в которую добавляется несколько объектов:

```
Vectors route = new Vectors();
route.Add(new Vector(2.0, 90.0));
route.Add(new Vector(1.0, 180.0));
route.Add(new Vector(0.5, 45.0));
route.Add(new Vector(2.5, 315.0));
```

Фрагмент кода ***Ch12Ex03\Program.cs***

Далее с помощью метода `Vectors.Sum()` элементы коллекции выводятся на консоль, как отмечалось ранее, на этот раз в первоначальном порядке:

```
Console.WriteLine(route.Sum());
```

Затем создается первый из делегатов — `sorter`.

Этот делегат имеет тип `Comparison<Vector>` и, следовательно, допускает присваивание ему метода со следующим возвращаемым типом и параметрами:

```
int метод(Vector objectA, Vector objectB)
```

Эта сигнатура совпадает с сигнатурой метода `VectorDelegates.Compare()`, который присваивается данному делегату:

```
Comparison<Vector> sorter = new Comparison <Vector>(VectorDelegates.Compare);
```

В методе `Compare()` производится сравнение абсолютных величин двух векторов следующим образом:

```
public static int Compare(Vector x, Vector y)
{
  if (x.R > y.R)
  {
    return 1;
  }
  else if (x.R < y.R)
  {
    return -1;
  }
  return 0;
}
```

Это позволяет упорядочивать векторы по их абсолютной величине:

```
route.Sort(sorter);
Console.WriteLine(route.Sum());
```

В выводе приложения результат суммирования выглядит, как и следовало ожидать, точно так же, потому что конечная точка следующего "векторного маршрута" остается той же, независимо от порядка, в котором выполняются отдельные шаги.

Далее с применением поиска строится подмножество векторов в коллекции. Для этого используется метод `VectorDelegates.TopRightQuadrant()`:

```
public static bool TopRightQuadrant(Vector target)
{
  if (target.Theta >= 0.0 && target.Theta <= 90.0)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Этот метод возвращает `true`, если значение `Theta`, передаваемое ему в аргументе `Vector`, лежит в диапазоне от 0 до 90 градусов, т.е. если вектор в диаграмме (вроде показанной ранее) указывает вверх и/или вправо.

В методе `Main()` данный метод используется через делегат типа `Predicate<Vector>`, как показано ниже:

```
Predicate<Vector> searcher =
  new Predicate<Vector>(VectorDelegates.TopRightQuadrant);
Vectors topRightQuadrantRoute = new Vectors(route.FindAll(searcher));
Console.WriteLine(topRightQuadrantRoute.Sum());
```

Это требует определения в `Vectors` соответствующего конструктора:

```
public Vectors(IEnumerable<Vector> initialItems)
{
  foreach (Vector vector in initialItems)
  {
    Add(vector);
  }
}
```

Здесь новая коллекция инициализируется с применением интерфейса `IEnumerable<Vector>`. Это необходимо потому, что `List<Vector>.FindAll()` возвращает экземпляр `List<Vector>`, а не экземпляр `Vectors`.

В результате выполнения поиска возвращается только подмножество объектов `Vector`, из-за чего (как и следовало ожидать) результат суммирования выглядит по-другому. Привыкание к использованию таких обобщенных типов делегатов для сортировки и поиска в обобщенных коллекциях может потребовать некоторого времени, но при таком подходе код получается более простым и эффективным с очень логичной структурой. Поэтому стоит уделить время на изучение описанных в настоящем разделе приемов.

В качестве дополнения к приведенному примеру следует отметить, что код

```
Comparison<Vector> sorter = new Comparison<Vector>(VectorDelegates.Compare);
route.Sort(sorter);
```

может быть упрощен, как показано ниже:

```
route.Sort(VectorDelegates.Compare);
```

Это устраняет необходимость в неявной ссылке на тип Comparison<Vector>. На самом деле экземпляр этого типа все равно создается, но создается он неявно. Очевидно, что экземпляр этого типа необходим методу Sort() для выполнения его работы, но компилятор осознает это и автоматически создает его в предоставляемом методе. В такой ситуации ссылка на метод VectorDelegates.Compare() (без круглых скобок) называется *группой методов*. Во многих ситуациях такие группы методов для неявного создания делегатов улучшают читабельность кода.

Тип Dictionary<K,V>

Тип Dictionary<K,V> позволяет определять коллекцию пар "ключ-значение". В отличие от других обобщенных типов-коллекций, которые уже рассматривались в настоящей главе, данный класс требует создания экземпляров двух типов — для ключа и для значения, которые представляют каждый элемент коллекции.

После создания над экземпляром Dictionary<K,V> можно выполнять в основном все те же самые операции, что и над экземпляром любого другого класса, унаследованного от DictionaryBase, но только с помощью поддерживаемых им методов и свойств, безопасных в отношении типов. Например, можно добавлять в него пары "ключ-значение" с помощью строго типизированного метода Add():

```
Dictionary<string, int> things = new Dictionary<string, int>();
things.Add("Green Things", 29);
things.Add("Blue Things", 94);
things.Add("Yellow Things", 34);
things.Add("Red Things", 52);
things.Add("Brown Things", 27);
```

Итерация по ключам и значениям в коллекции осуществляется с использованием свойств Keys и Values:

```
foreach (string key in things.Keys)
{
  Console.WriteLine(key);
}

foreach (int value in things.Values)
{
  Console.WriteLine(value);
}
```

По элементам коллекции можно также проходить за счет получения каждого из них в виде экземпляра KeyValuePair<K,V>, во многом подобно тому, как это делалось с объектами DictionaryEntry, которые демонстрировались в предыдущей главе:

```
foreach (KeyValuePair<string, int> thing in things)
{
  Console.WriteLine("{0} = {1}", thing.Key, thing.Value);
}
```

Обратите внимание, что в Dictionary<K,V> ключи элементов должны быть уникальными. Попытка добавить элемент с ключом, совпадающим с ключом существующего элемента, приводит к генерации исключения ArgumentException. По этой причине конструктору Dictionary<K,V> допускается передавать в качестве аргумента интерфейс IComparer<K>. Это может быть необходимо, когда в качестве ключей используются собственные классы, которые не поддерживают интерфейс IComparable или IComparable<K>, либо когда объекты должны сравниваться посредством какого-то нестандартного процесса. Например, в предыдущем примере для сравнения строковых ключей можно было бы применять метод, не учитывающий регистр символов:

```
Dictionary<string, int> things =
  new Dictionary<string, int>(StringComparer.CurrentCultureIgnoreCase);
```

Тогда для приведенных ниже ключей будет генерироваться исключение:

```
things.Add("Green Things", 29);
things.Add("Green things", 94);
```

Конструктору `Dictionary<K,V>` можно также передать в качестве аргумента значение начальной емкости (в виде `int`) или список элементов (с применением интерфейса `IDictionary<K,V>`).

Модификация проекта `CardLib` для использования класса обобщенной коллекции

Теперь можно внести в проект `CardLib`, который разрабатывается на протяжении нескольких последних глав, одно простое изменение, а именно — модифицировать класс `Cards` так, чтобы в нем использовался обобщенный класс коллекции; это позволит сократить количество требуемых строк кода. Ниже показано обновленное определение класса `Cards`:

```
public class Cards : List<Card>, ICloneable
{
...
}
```

Фрагмент кода ***Ch12CardLib\Cards.cs***

Можно также удалить из класса `Cards` все методы, кроме `Clone()`, который необходимо для `ICloneable`, и `CopyTo()`, потому что версия `CopyTo()`, предоставляемая `List<Card>`, работает с массивом объектов `Card`, а не коллекцией `Cards`. Метод `Clone()` требует небольшой модификации, поскольку в классе `List<T>` не определено свойство `List`:

```
public object Clone()
{
  Cards newCards = new Cards();
  foreach (Card sourceCard in this)
  {
    newCards.Add(sourceCard.Clone() as Card);
  }
  return newCards;
}
```

Полный код проекта `Ch12CardLib` входит в состав кода примеров для настоящей главы.

Определение обобщенных типов

К этому моменту представлено достаточно сведений об обобщениях, чтобы приступить к их созданию. Было показано много примеров кода с участием обобщенных типов. В этом разделе речь пойдет о том, как определять следующее:

- обобщенные классы;
- обобщенные интерфейсы;
- обобщенные методы;
- обобщенные делегаты.

Кроме того, здесь будут продемонстрированы более сложные темы, связанные с определением обобщенных типов:

- ключевое слово `default`;
- ограничение типов;
- наследование от обобщенных классов;
- обобщенные операции.

Определение обобщенных классов

Для создания обобщенного класса просто добавьте к определению класса угловые скобки:

```
class MyGenericClass<T>
{
  ...
}
```

На месте `T` может быть указан любой идентификатор, соответствующий стандартным правилам именования C#. Однако обычно используется просто буква `T`. Определение обобщенного класса может содержать любое количество параметров, представляющих типы; они отделяются друг от друга запятыми:

```
class MyGenericClass<T1, T2, T3>
{
  ...
}
```

После определения типы можно использовать в определении обобщенного класса точно так же, как любые другие типы. Их можно применять в качестве типов для переменных-членов, возвращаемых типов для таких членов, как свойства или методы, и типов для аргументов методов:

```
class MyGenericClass<T1, T2, T3>
{
  private T1 innerT1Object;
  public MyGenericClass(T1 item)
 {
    innerT1Object = item;
  }
  public T1 InnerT1Object
  {
    get
    {
      return innerT1Object;
    }
 }
}
```

Здесь конструктору может передаваться объект типа `T1`, и к нему возможен доступ только для чтения через свойство `InnerT1Object`. Обратите внимание, что практически никаких предположений касательно того, какими являются предоставляемые классу типы, делать нельзя. Следующий код, например, компилироваться не будет:

```
class MyGenericClass <T1, T2, T3>
{
  private T1 innerT1Object;
  public MyGenericClass()
  {
    innerT1Object = new T1();
  }
  public T1 InnerT1Object
  {
```

```
    get
    {
      return innerT1Object;
    }
  }
}
```

Поскольку тип T1 не известен, использовать какой бы то ни было из его конструкторов нельзя — он может вообще не иметь конструктора или не располагать общедоступным конструктором по умолчанию. Без написания более сложного кода, предусматривающего применение приемов, которые будут показаны позже в этом разделе, о типе T1, по сути, можно делать только следующее предположение: он либо унаследован от System.Object, либо может быть в него упакован.

Очевидно, это означает, что ничего особого интересного с экземплярами данного типа и любого из других типов, которые предоставляются обобщенному классу MyGenericClass, делать невозможно. Без применения *рефлексии*, которая представляет собой усовершенствованный прием для изучения типов во время выполнения (в этой главе он не рассматривается), возможным будет только код, не сложнее показанного ниже:

```
public string GetAllTypesAsString()
{
  return "T1 = " + typeof(T1).ToString()
    + ", T2 = " + typeof(T2).ToString()
    + ", T3 = " + typeof(T3).ToString();
}
```

Свыше этого мало что можно сделать, особенно в случае коллекций, поскольку работа с группами объектов — довольно простой процесс, который не нуждается в каких-либо предположениях о типе объектов. Это является одной из веских причин, по которым существуют классы обобщенных коллекций, рассмотренные ранее в настоящей главе.

Другое ограничение заключается в том, что при сравнении значения типа, предоставляемого в обобщенном классе, с null, разрешено пользоваться только операцией == или !=. То есть следующий код будет работать нормально:

```
public bool Compare(T1 op1, T1 op2)
{
  if (op1 != null && op2 != null)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Если T1 является типом значения, то всегда предполагается, что он не может быть null, и потому Compare будет всегда возвращаться true. Однако при попытке сравнить два аргумента op1 и op2 компилятор будет сообщать об ошибке:

```
public bool Compare(T1 op1, T1 op2)
{
  if (op1 == op2)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Причина в том, что в данном коде предполагается, что T1 поддерживает операцию ==. Короче говоря, чтобы делать нечто по-настоящему интересное с обобщениями, необходимо знать немного больше о типах, которые используются в классе.

Ключевое слово default

Одной из основных деталей, которые обычно необходимо знать о типах, используемых для создания экземпляров обобщенных классов — являются они ссылочными или типами значения. Не зная этого, невозможно даже присваивать им null с помощью такого кода, как показан ниже:

```
public MyGenericClass()
{
  innerT1Object = null;
}
```

Если T1 представляет собой тип значения, тогда innerT1Object не может иметь значение null, поэтому код не скомпилируется. К счастью, эта проблема была изучена и привела к появлению нового способа для использования ключевого слова default (применение которого уже демонстрировалось в структурах switch ранее в книге):

```
public MyGenericClass()
{
  innerT1Object = default(T1);
}
```

В случае если T1 является ссылочным типом, innerT1Object будет присваиваться null, а если типом значения — значение по умолчанию. Таким значением по умолчанию для числовых типов является 0, а у структур каждый из их членов аналогичным образом инициализируется в 0 или null. Ключевое слово default позволяет выполнять с назначаемыми обязательными типами немного больше манипуляций, но чтобы двинуться еще дальше, необходимо научиться ограничивать поставляемые типы.

Ограничение типов

Типы, которые использовались с обобщенными классами до сих пор, называются *неограниченными*, поскольку никаких ограничений на то, какими они могут быть, на них не накладывалось. С помощью *ограничения* типов можно сужать перечень типов, которые могут использоваться для создания экземпляра обобщенного класса. Делать это можно несколькими способами. Например, можно ограничивать тип только тем, который наследуется от определенного типа. Если взять приводившиеся ранее классы Animal, Cow и Chicken, то в их случае тип можно было бы ограничить только унаследованным от Animal, и тогда следующий код работал бы нормально:

```
MyGenericClass<Cow> = new MyGenericClass<Cow>();
```

А вот показанный ниже код приводил бы к выдаче во время компиляции сообщения об ошибке:

```
MyGenericClass<string> = new MyGenericClass<string>();
```

Добиться подобного поведения можно за счет использования в определениях своих классов ключевого слова where:

```
class MyGenericClass<T> where T : ограничение
{
  ...
}
```

Здесь на месте *ограничение* указывается накладываемое ограничение. Подобным образом можно задавать сразу несколько ограничений, отделяя их друг от друга запятыми:

```
class MyGenericClass<T> where T : ограничение1, ограничение2
{
  ...
}
```

Определять ограничения можно как для одного, так и для всех необходимых обобщенному классу типов, указывая соответствующее количество операторов `where`:

```
class MyGenericClass <T1, T2> where T1 : ограничение1 where T2 : ограничение2
{
  ...
}
```

Ограничения должны идти после спецификаторов наследования:

```
class MyGenericClass <T1, T2> : MyBaseClass, IMyInterface
where T1 : ограничение1 where T2 : ограничение2
{
...
}
```

Доступные варианты ограничений перечислены в табл. 12.5.

Таблица 12.5. Доступные ограничения

Ограничение	Определение	Пример использования
`struct`	Тип должен быть типом значения	В классе, которому для работы требуются типы значения, например, в классе, где значение `0` переменной экземпляра типа `T` имеет какой-нибудь смысл
`class`	Тип должен быть ссылочным типом	В классе, которому для работы требуются ссылочные типы, например, в классе, где значение `null` переменной типа `T` имеет какой-нибудь смысл
`base-class`	Тип должен быть `base-class` либо наследоваться от него. В этом ограничении можно указывать любое имя класса	В классе, которому для работы требуется определенная базовая функциональность, унаследованная от `base-class`
`interface`	Тип должен быть либо реализовать `interface`	В классе, которому для работы требуется определенная базовая функциональность, предоставляемая `interface`
`new()`	Тип должен иметь общедоступный конструктор без параметров	В классе, где необходимо иметь возможность создавать переменные экземпляра типа `T`, к примеру, в конструкторе

Ограничение `new()` должно указываться для типа последним.

Допускается применять параметр одного типа в качестве ограничения для другого с использованием ограничения `base-class` следующим образом:

```
class MyGenericClass <T1, T2> where T2 : T1
{
  ...
}
```

Здесь тип T2 должен быть того же типа, что и T1, либо наследоваться от него. Такое ограничение называется *простым ограничением типа* и означает, что параметр одного обобщенного типа применяется в качестве ограничения для другого.

Использовать циклические ограничения типов, вроде показанных ниже, не разрешено:

```
class MyGenericClass <T1, T2> where T2 : T1 where T1 : T2
{
  ...
}
```

Этот код компилироваться не будет. В следующем практическом занятии будет определен обобщенный класс, который использует классы из семейства Animal, показанного в предыдущих главах.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Определение обобщенного класса

1. Создайте новое консольное приложение по имени Ch12Ex04 и сохраните его в каталоге C:\BegVCSharp\Chapter12.
2. Щелкните правой кнопкой мыши на имени этого проекта в окне Solution Explorer и выберите в контекстном меню пункт Add⇨Existing Item (Добавить⇨Существующий элемент).
3. Выберите из каталога C:\BegVCSharp\Chapter12\Ch12Ex02\Ch12Ex02 файлы Animal.cs, Cow.cs, Chicken.cs, SuperCow.cs и Farm.cs и щелкните на кнопке Add (Добавить).
4. Измените объявление пространства имен в каждом из добавленных файлов следующим образом:

```
namespace Ch12Ex04
```

*Фрагмент кода **Ch12Ex04\Animal.cs, Ch12Ex04\Cow.cs и Ch12Ex04\Chicken.cs***

5. Модифицируйте код в файле Animal.cs, как показано ниже:

```
public abstract class Animal
{
  ...
  public abstract void MakeANoise();
}
```

*Фрагмент кода **Ch12Ex04\Animal.cs***

6. Измените код в файле Chicken.cs следующим образом:

```
public class Chicken : Animal
{
  ...
  public override void MakeANoise()
  {
    Console.WriteLine("{0} says 'cluck!'", name);
                    // кудахчет
  }
}
```

*Фрагмент кода **Ch12Ex04\Chicken.cs***

7. Модифицируйте код в файле Cow.cs, как показано ниже:

```
public class Cow : Animal
{
  ...
  public override void MakeANoise()
  {
    Console.WriteLine("{0} says 'moo!'", name);
                    // мычит
  }
}
```

Фрагмент кода ***Ch12Ex04\Cow.cs***

8. Добавьте новый класс с именем SuperCow и измените код в файле SuperCow.cs следующим образом:

```
public class SuperCow : Cow
{
  public void Fly()
  {
    Console.WriteLine("{0} is flying!", name);
                    // летит
  }
  public SuperCow(string newName) : base(newName)
  {
  }
  public override void MakeANoise()
  {
    Console.WriteLine("{0} says 'here I come to save the day!'", name);
                    // говорит, что пришла спасти положение
  }
}
```

Фрагмент кода ***Ch12Ex04\SuperCow.cs***

9. Добавьте новый класс по имени Farm и измените код в файле Farm.cs следующим образом:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch12Ex04
{
  public class Farm<T> : IEnumerable<T>
  where T : Animal
 {
  private List<T> animals = new List<T>();
  public List<T> Animals
  {
    get
    {
      return animals;
    }
  }

  public IEnumerator<T> GetEnumerator()
  {
    return animals.GetEnumerator();
  }
```

```
  IEnumerator IEnumerable.GetEnumerator()
  {
    return animals.GetEnumerator();
  }
  public void MakeNoises()
  {
    foreach (T animal in animals)
    {
    animal.MakeANoise();
  }
}
public void FeedTheAnimals()
 {
   foreach (T animal in animals)
   {
     animal.Feed();
   }
  }
   public Farm<Cow> GetCows()
   {
     Farm<Cow> cowFarm = new Farm<Cow>();
     foreach (T animal in animals)
     {
       if (animal is Cow)
       {
         cowFarm.Animals.Add(animal as Cow);
       }
     }
     return cowFarm;
   }
  }
}
```

Фрагмент кода ***Ch12Ex04\Farm.cs***

10. Модифицируйте код в файле `Program.cs`, как показано ниже:

```
static void Main(string[] args)
{
  Farm<Animal> farm = new Farm<Animal>();
  farm.Animals.Add(new Cow("Jack"));
  farm.Animals.Add(new Chicken("Vera"));
  farm.Animals.Add(new Chicken("Sally"));
  farm.Animals.Add(new SuperCow("Kevin"));
  farm.MakeNoises();
  Farm<Cow> dairyFarm = farm.GetCows();
  dairyFarm.FeedTheAnimals();

  foreach (Cow cow in dairyFarm)
  {
    if (cow is SuperCow)
    {
      (cow as SuperCow).Fly();
    }
  }
  Console.ReadKey();
}
```

Фрагмент кода ***Ch12Ex04\Program.cs***

11. Запустите приложение. На рис. 12.5 показан результат, который должен получиться.

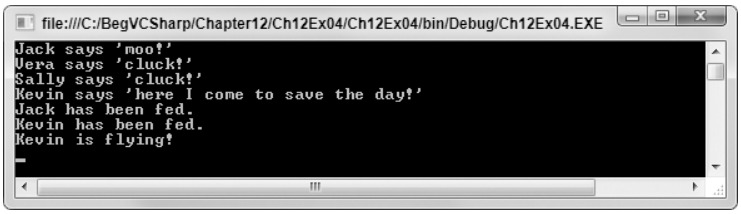


Рис. 12.5. *Приложение* `Ch12Ex04` *в действии*

Описание работы

В этом примере был создан обобщенный класс по имени `Farm<T>`, который вместо того, чтобы наследоваться от обобщенного класса списка, предоставляет его в виде общедоступного свойства. За тип этого списка отвечает параметр `T`, который передается `Farm<T>` и ограничивается типом, который либо является `Animal` либо унаследован от него:

```
public class Farm<T> : IEnumerable<T>
  where T : Animal
{
  private List<T> animals = new List<T>();

  public List<T> Animals
  {
    get
    {
      return animals;
    }
  }
```

Фрагмент кода ***Ch12Ex04\Farm.cs***

В `Farm<T>` также реализован интерфейс `IEnumerable<T>`, которому тоже передается параметр `T` с теми же ограничениями. Реализуется этот интерфейс для обеспечения возможности итерации по содержащимся в `Farm<T>` элементам без явного прохода по `Farm<T>.Animals`. Достигнуть этого очень легко: нужно просто вернуть перечислитель, который предоставляется в `Animals` и роль которого там исполняет класс `List<L>`, также реализующий интерфейс `IEnumerable<T>`:

```
public IEnumerator <T> GetEnumerator()
{
  return animals.GetEnumerator();
}
```

Так как `IEnumerable<T>` унаследован от `IEnumerable`, в `Farm<T>` реализован и метод `IEnumerable.GetEnumerator()`:

```
IEnumerator IEnumerable.GetEnumerator()
{
  return animals.GetEnumerator();
}
```

Далее в `Farm<T>` создаются два новых метода, которые используют методы из абстрактного класса `Animal`:

```
public void MakeNoises()
{
  foreach (T animal in animals)
  {
    animal.MakeANoise();
  }
}
```

```
public void FeedTheAnimals()
{
  foreach (T animal in animals)
  {
    animal.Feed();
  }
}
```

Из-за того, что T ограничивается типом Animal, данный код компилируется нормально — и доступ к этим методам гарантирован, каким бы на самом деле не оказался тип T.

Следующий метод — GetCows() — более интересен. В нем просто извлекаются из коллекции все элементы, относящиеся к типу Cow (или унаследованы от Cow, как, например, новый класс SuperCow):

```
public Farm<Cow> GetCows()
{
  Farm<Cow> cowFarm = new Farm<Cow>();
  foreach (T animal in animals)
  {
    if (animal is Cow)
    {
      cowFarm.Animals.Add(animal as Cow);
    }
  }
  return cowFarm;
}
```

Интересно здесь то, что данный метод выглядит несколько неэкономным. Если нужно использовать другие методы такого рода, например, GetChickens() и т.п., их также понадобилось бы реализовать явно. В системе с большим количеством типов таких методов было бы еще больше. Поэтому гораздо эффективнее здесь предусмотреть *обобщенный метод*, который будет реализован немного позже в главе.

В клиентском коде внутри Program.cs выполняется лишь тестирование различных методов Farm и не содержится ничего такого, о чем бы ни рассказывалось ранее.

Наследование от обобщенных классов

Класс Farm<T> в предыдущем примере, а также ряд других классов, которые демонстрировались ранее в главе, наследовались от обобщенного типа. В случае Farm<T> этим типом был интерфейс IEnumerable<T>. Ограничение, накладываемое на тип T в Farm<T>, приводило к накладыванию соответствующего дополнительного ограничения и на тип T, используемый в IEnumerable<T>. Подобный подход может быть очень удобным для установки ограничений на никак не ограниченные типы. Однако при этом должны соблюдаться некоторые правила.

Нельзя "снимать" с типов ограничения, которые накладываются в типе, от которого они унаследованы. Другими словами, тип T, используемый в типе, от которого осуществляется наследование, должен обязательно ограничиваться как минимум настолько же сильно, насколько и этот тип. Например, следующий код является допустимым:

```
class SuperFarm<T> : Farm<T>
     where T : SuperCow
{
}
```

Этот код будет работать, потому что T ограничивается типом Animal в классе Farm<T>, а ограничение этого класса до класса SuperCow означает ограничение T до подмножества именно таких значений. Тем не менее, показанный ниже код компилироваться не будет:

```
class SuperFarm<T> : Farm<T>
     where T : struct
{
}
```

Здесь точно понятно, что тип T, предоставляемый SuperFarm<T>, не может быть преобразован в тип T, пригодный для использования в Farm<T>, поэтому код и не компилируется.

Даже в ситуациях, когда ограничение является надмножеством, все равно возникает та же проблема:

```
class SuperFarm<T> : Farm<T>
     where T : class
{
}
```

Хотя типы вроде Animal разрешено использовать в SuperFarm<T>, другие типы, удовлетворяющие ограничению class, все равно будут недопустимыми в Farm<T>. По этой причине данный код компилироваться не будет. Это правило касается абсолютно всех типов ограничений, которые демонстрировались ранее в настоящей главе.

Обратите внимание, что при наследовании от обобщенного типа должна быть предоставлена вся необходимая информация о типах либо в виде параметров других обобщенных типов, как показывалось выше, либо явно. То же касается и необобщенных классов, унаследованных от обобщенных типов, которые демонстрировались в других местах. Ниже приведен пример:

```
public class Cards : List<Card>, ICloneable
{
}
```

Этот код будет компилироваться, а вот следующий — нет:

```
public class Cards : List<T>, ICloneable
{
}
```

Здесь не предоставлена информация для T, поэтому компилятор сообщит об ошибке.

Если обобщенному типу предоставлен параметр, как было в List<Card> выше, он может называться закрытым (closed) обобщенным типом. Аналогично, наследование от List<T> может быть названо наследованием от открытого (open) обобщенного типа.

Обобщенные операции

Переопределенные операции реализуются в C# подобно другим методам и, следовательно, могут быть реализованы и в обобщенных классах. Например, в Farm<T> можно определить следующую операцию неявного преобразования:

```
public static implicit operator List<Animal>(Farm<T> farm)
{
  List<Animal> result = new List<Animal>();
  foreach (T animal in farm)
  {
    result.Add(animal);
  }
  return result;
}
```

Это позволит при необходимости получать доступ к объектам Animal в Farm<T> напрямую как к List<Animal>.

Это очень полезно для реализации сложения двух экземпляров Farm<T>, например, с помощью показанных ниже операций:

```
public static Farm<T> operator +(Farm<T> farm1, List<T> farm2)
{
  Farm<T> result = new Farm<T>();
  foreach (T animal in farm1)
  {
    result.Animals.Add(animal);
  }
  foreach (T animal in farm2)
  {
   if (!result.Animals.Contains(animal))
   {
     result.Animals.Add(animal);
   }
  }
  return result;
}
public static Farm<T> operator +(List<T> farm1, Farm<T> farm2)
{
  return farm2 + farm1;
}
```

Тогда складывать экземпляры Farm<Animal> и Farm<Cow> можно следующим образом:

```
Farm<Animal> newFarm = farm + dairyFarm;
```

В этом коде dairyFarm (экземпляр Farm<Cow>) неявно преобразуется в экземпляр List<Animal>, пригодный для использования в перегруженной операции + внутри Farm<T>.

Может показаться, что подобного можно добиться и посредством такого кода:

```
public static Farm<T> operator +(Farm<T> farm1, Farm<T> farm2)
{
  Farm<T> result = new Farm<T>();
  foreach (T animal in farm1)
  {
    result.Animals.Add(animal);
  }
  foreach (T animal in farm2)
  {
    if (!result.Animals.Contains(animal))
    {
      result.Animals.Add(animal);
    }
  }
  return result;
}
```

Однако из-за того, что Farm<Cow> не может быть преобразован в Farm<Animal>, операция суммирования завершается ошибкой. Чтобы устранить проблему, можно реализовать следующую операцию преобразования:

```
public static implicit operator Farm<Animal>(Farm<T> farm)
{
  Farm<Animal> result = new Farm<Animal>();
  foreach (T animal in farm)
  {
    result.Animals.Add(animal);
  }
  return result;
}
```

С помощью этой операции экземпляры Farm<T>, такие как Farm<Cow>, смогут преобразовываться в экземпляры Farm<Animal> и проблема, следовательно, исчезнет. Применять допускается любой из этих подходов, хотя последний, несомненно, более предпочтителен в силу своей простоты.

Обобщенные структуры

В предыдущих главах уже рассказывалось о том, что структуры выглядят, по сути, так же, как классы, а отличаются от них лишь незначительными деталями и тем, что представляют собой не ссылочные типы, а типы значения. В результате создавать *обобщенные структуры* можно тем же образом, что и обобщенные классы:

```
public struct MyStruct<T1, T2>
{
  public T1 item1;
  public T2 item2;
}
```

Определение обобщенных интерфейсов

Применение нескольких обобщенных интерфейсов уже демонстрировалось в этой главе: это были интерфейсы из пространства имен Systems.Collections.Generic, такие как IEnumerable<T>, который использовался в последнем примере. При определении обобщенных интерфейсов применяются те же самые приемы, что и для обобщенных классов:

```
interface MyFarmingInterface<T>
  where T : Animal
{
  bool AttemptToBreed(T animal1, T animal2);

  T OldestInHerd
  {
    get;
  }
}
```

Здесь обобщенный параметр T служит для указания типа в двух аргументах метода AttemptToBreed() и в свойстве OldestInHerd.

Правила наследования применяются те же, что и для классов. При наследовании от базового обобщенного интерфейса должны соблюдаться правила, такие как сохранение ограничений, установленных для параметров обобщенных типов в базовом интерфейсе.

Определение обобщенных методов

В последнем практическом занятии использовался метод по имени GetCows(), в пояснениях к которому упоминалось о возможности создания более универсальной его версии за счет применения так называемого *обобщенного метода*. В этом разделе будут описаны необходимые детали. Обобщенным называется такой метод, в котором возвращаемый тип и/или типы параметров определяются параметром или параметрами обобщенного типа:

```
public T GetDefault<T>()
{
  return default(T);
}
```

В этом типичном примере с помощью рассматривавшегося ранее ключевого слова default возвращается значение по умолчанию для типа T. Вызывается этот метод следующим образом:

```
int myDefaultInt = GetDefault<T>();
```

Отвечающий за тип параметр T предоставляется во время вызова метода.

Этот тип T довольно отличается от типов, используемых для передачи классам параметров обобщенного типа. В действительности обобщенные методы могут быть реализованы в не обобщенных классах:

```
public class Defaulter
{
  public T GetDefault<T>()
  {
   return default(T);
  }
}
```

Если класс является обобщенным, то для типов обобщенного метода должны выбираться разные идентификаторы. Например, следующий код компилироваться не будет:

```
public class Defaulter<T>
{
  public T GetDefault<T>()
  {
    return default(T);
  }
}
```

Здесь тип T необходимо переименовать — либо в методе, либо в классе.

Ограничения для параметров обобщенного метода используются тем же образом, что и для классов, и в этом случае можно применять любые параметры типа класса:

```
public class Defaulter<T1>
{
  public T2 GetDefault<T2>()
    where T2 : T1
  {
    return default(T2);
  }
}
```

Здесь тип T2, предоставляемый методу, должен обязательно быть тем же самым либо наследоваться от типа T1, предоставляемого классу. Такой прием наиболее часто применяется для наложения ограничений на обобщенные методы.

В показанный выше класс Farm<T> может быть включен следующий метод (который присутствует в проекте Ch12Ex04, но помещен в комментарии):

```
public Farm<U> GetSpecies<U>() where U : T
{
  Farm<U> speciesFarm = new Farm<U>();
  foreach (T animal in animals)
  {
    if (animal is U)
    {
      speciesFarm.Animals.Add(animal as U);
    }
  }
  return speciesFarm;
}
```

Это позволит заменить метод GetCows() и любые другие методы того же типа. Используемый здесь параметр обобщенного типа — U — ограничивается параметром T, на который, в свою очередь накладывает ограничения класс Farm<T>, который разрешает использовать для него только тип Animal. Это предоставляет возможность обрабатывать экземпляры T как экземпляры Animal, если вдруг возникнет такое желание.

Применение этого нового метода требует внесения в клиентский код в Program.cs проекта Ch12Ex04 следующего изменения:

```
Farm<Cow> dairyFarm = farm.GetSpecies<Cow>();
```

То же самое можно записать и по-другому:

```
Farm<Chicken> dairyFarm = farm.GetSpecies<Chicken>();
```

Можно также использовать любой другой класс, унаследованный от Animal.

Обратите внимание, что наличие параметров обобщенного типа в методе приводит к изменению его сигнатуры. Это означает, что может существовать несколько перегрузок метода, отличающихся друг от друга только параметрами обобщенного типа, что демонстрируется в следующем примере:

```
public void ProcessT<T>(T op1)
{
  ...
}
public void ProcessT<T, U>(T op1)
{
  ...
}
```

На выбор метода для использования влияет количество указанных при вызове метода параметров обобщенного типа.

Определение обобщенных делегатов

Последним обобщенным типом, который осталось рассмотреть, является *обобщенный делегат*. Обобщенные делегаты в действии вы уже видели ранее в этой главе при рассмотрении способов выполнения сортировки и поиска в обобщенных списках, где для того и другого применялись, соответственно, делегаты Comparison<T> и Predicate<T>.

В главе 6 было показано, как определять делегаты за счет указания параметров и возвращаемого типа метода, а также ключевого слова delegate и имени делегата:

```
public delegate int MyDelegate(int op1, int op2);
```

Для определения обобщенного делегата достаточно просто объявить его и использовать один или более параметров обобщенного типа:

```
public delegate T1 MyDelegate<T1, T2>(T2 op1, T2 op2) where T1 : T2;
```

Как здесь видно, в случае обобщенных делегатов также можно накладывать ограничения. В следующей главе делегаты рассматриваются более подробно, в том числе их применение в событиях — очень часто встречающий во время программирования на C# прием.

Вариантность

Вариантность (variance) представляет собой собирательный термин для обозначения *ковариантности* (covariance) и *контравариантности* (contravariance) — двух концепций, появившихся в .NET 4. На самом деле они существовали уже давно (и были доступные еще в .NET 2.0), но до выхода версии .NET 4.0 их было очень трудно реализовать, потому что для этого требовалось применять специальные процедуры компиляции.

Проще всего понять, что собой представляют эти понятия — сравнить их с полиморфизмом. Полиморфизм, как известно, представляет собой технологию, которая позволяет помещать объекты производного типа в переменные базового типа, например:

```
Cow myCow = new Cow("Geronimo");
Animal myAnimal = myCow;
```

Здесь объект типа Cow был помещен в переменную типа Animal, что возможно потому, что Cow унаследован от Animal.

Однако с интерфейсами подобное делать нельзя. То есть следующий код работать не будет:

```
IMethaneProducer<Cow> cowMethaneProducer = myCow;
IMethaneProducer<Animal> animalMethaneProducer = cowMethaneProducer;
```

Здесь первая строка кода вполне допустима при условии, что Cow поддерживает интерфейс IMethaneProducer<Cow>. Тем не менее, во второй строке предполагается наличие отношения между двумя типами интерфейсов, которого не существует, и потому нет способа выполнить преобразование одного в другой. Или все-таки возможность есть? Описанные до этого момента приемы такого, конечно же, не позволяют, поскольку все параметры обобщенных типов являются *инвариантными* (invariant). Однако для обобщенных интерфейсов и обобщенных делегатов можно определять вариантные параметры типа, что как раз и подходит для показанной в предыдущем коде ситуации.

Чтобы заставить работать предыдущий код, параметр типа T для интерфейса IMethaneProducer<T> должен быть *ковариантным*. Наличие ковариантного параметра типа, по сути, устанавливает отношение наследования между IMethaneProducer<Cow> и IMethaneProducer<Animal>, так что переменные одного типа могут содержать значения другого типа, точно так же как и в случае полиморфизма (хотя и несколько более сложным образом).

Чтобы завершить знакомство с вариантностью, необходимо рассмотреть другую ее разновидность — *контравариантность*. Она выглядит похоже, но работает в другом направлении. Вместо того чтобы давать возможность помещать значение обобщенного интерфейса в переменную, которая включает базовый тип, как при ковариантности, она позволяет помещать этот интерфейс в переменную, в которой используется производный тип, например:

```
IGrassMuncher<Cow> cowGrassMuncher = myCow;
IGrassMuncher<SuperCow> superCowGrassMuncher = cowGrassMuncher;
```

На первый взгляд этот код выглядит немного странно, но добиться такого же с помощью полиморфизма не получилось бы. Как будет показано в разделе "Контравариантность", при определенных обстоятельствах этот прием оказывается довольно полезным.

Два следующих раздела посвящены самостоятельной реализации вариантности в обобщенных типах и применению вариантности в .NET Framework для облегчения жизни разработчикам.

Весь приведенный код доступен в виде демонстрационного проекта VarianceDemo.

Ковариантность

Для определения параметра обобщенного типа как ковариантного необходимо использовать в определении типа ключевое слово out, как показано в следующем примере:

```
public interface IMethaneProducer<out T>
{
...
}
```

В определениях интерфейсов параметры ковариантных типов могут применяться только в качестве возвращаемых значений из методов или get-блоков свойств.

Хорошей демонстрацией пользы от ковариантности является такой поставляемый в .NET Framework интерфейс, как IEnumerable<T>, который уже использовался ранее в этой главе. Тип T в этом интерфейсе определен как ковариантный. Это означает, что в переменную типа IEnumerable<Animal> можно помещать объект, поддерживающий, скажем, IEnumerable<Cow>:

```
static void Main(string[] args)
{
  List<Cow> cows = new List<Cow>();
  cows.Add(new Cow("Geronimo"));
  cows.Add(new SuperCow("Tonto"));
  ListAnimals(cows);
  Console.ReadKey();
}
static void ListAnimals(IEnumerable<Animal> animals)
{
  foreach (Animal animal in animals)
  {
    Console.WriteLine(animal.ToString());
  }
}
```

Здесь переменная `cows` относится к типу `List<Cow>` и поддерживает интерфейс `IEnumerable<Cow>`. Посредством ковариантности эта переменная может передаваться методу, который ожидает параметра типа `IEnumerable<Animal>`. Из того, что уже рассказывалось о работе циклов `foreach`, уже известно, что метод `GetEnumerator()` применятся для получения перечислителя `IEnumerator<T>`, а свойство `Current` этого перечислителя служит для доступа к элементам. В `IEnumerator<T>` также определяется ковариантным его параметр типа, а это означает, что его можно использовать в качестве `get`-блока параметра.

Контравариантность

Чтобы определить параметр обобщенного типа как контравариантный, необходимо указать в определении типа ключевое слово `in`:

```
public interface IGrassMuncher<in T>
{
...
}
```

В определениях интерфейсов параметры контравариантного типа можно использовать только в качестве параметров методов, но не возвращаемых типов.

И снова для лучшего понимания рассмотрим пример применения контравариантности в .NET Framework. Одним из интерфейсов, который имеет параметр контравариантного типа, является уже использовавшийся в главе интерфейс `IComparer<T>`. Его можно реализовать для животных (`Animal`) следующим образом:

```
public class AnimalNameLengthComparer : IComparer<Animal>
{
   public int Compare(Animal x, Animal y)
   {
      return x.Name.Length.CompareTo(y.Name.Length);
   }
}
```

Этот класс сравнивает животных по длине клички, благодаря чему его можно использовать для сортировки, например, экземпляра `List<Animal>`. Однако за счет контравариантности его можно также применять для сортировки экземпляра `List<Cow>`, даже несмотря на то, что метод `List<Cow>.Sort()` ожидает экземпляра `IComparer<Cow>`:

```
List<Cow> cows = new List<Cow>();
cows.Add(new Cow("Geronimo"));
cows.Add(new SuperCow("Tonto"));
cows.Add(new Cow("Gerald"));
cows.Add(new Cow("Phil"));
cows.Sort(new AnimalNameLengthComparer());
```

В большинстве обстоятельств контравариантность представляет собой нечто, что просто происходит, и она была встроена в .NET Framework для оказания помощи в решении задач подобного рода. В .NET Framework 4 имеется возможность реализовать оба типа вариантности самостоятельно, применяя описанные в этом разделе приемы.

Резюме

В этой главе было показано, как использовать стандартные обобщенные типы в C# и создавать собственные обобщенные типы, в том числе обобщенные классы, интерфейсы, методы и делегаты. Также рассматривалась работа со структурами, создание нулевых типов и применение классов из пространства имен `System.Collecitons.Generic`.

Вы узнали, что обобщения являются чрезвычайно мощной новой технологией в C#. С их помощью можно создавать классы, которые способны служить сразу нескольким целям и подходят для использования во множестве разнообразных ситуаций. Даже если нет причин создавать собственные обобщенные типы, наверняка придется часто работать со стандартными обобщенными классами коллекций.

В следующей главе мы продолжим изучение основных аспектов языка C# и рассмотрим события.

Упражнения

1. Какие из следующих элементов могут быть обобщенными?
   а) Классы
   б) Методы
   в) Свойства
   г) Перегруженные версии операций
   д) Структуры
   е) Перечисления
2. Расширьте класс `Vector` из примера `Ch12Ex01` так, чтобы операция `*` в нем предусматривала возвpaп скалярного произведения двух векторов.

Под скалярным произведением двух векторов понимается произведение их абсолютных величин, умноженное на косинус угла между ними.

3. Что неверно в следующем коде? Устраните ошибку.

```
public class Instantiator<T>
{
  public T instance;

   public Instantiator()
  {
    instance = new T();
  }
}
```

4. Что неверно в следующем коде? Устраните ошибку.

```
public class StringGetter<T>
{
  public string GetString<T>(T item)
  {
    return item.ToString();
  }
}
```

5. Создайте обобщенный класс по имени `ShortCollection<T>`, который реализует `IList<T>` и включает в себя коллекцию элементов с максимальным размером. Этот максимальный размер должен иметь вид целого числа, которое мог бы принимать конструктор `ShortCollection<T>`, и которое по умолчанию могло бы устанавливаться в 10. Конструктор также должен быть способен принимать начальный список элементов в параметре `List<T>`. Класс должен функционировать подобно `Collection<T>`, но генерировать исключение типа `IndexOutOfRangeException`, если производится попытка добавить в коллекцию слишком много элементов или если в передаваемом конструктору списке `List<T>` обнаруживается слишком большое количество элементов.

6. Будет ли компилироваться следующий код? Если нет, то почему?

```
public interface IMethaneProducer<out T>
{
   void BelchAt(T target);
}
```

Решения упражнений приведены в приложении A.

Что вы узнали в этой главе

Тема	Основные концепции
Использование обобщенных типов	Обобщенные типы требуют для своей работы одного или более параметров типов. Обобщенный тип можно использовать в качестве типа переменной, передавая ему необходимые параметры типов во время объявления переменной. Параметры типов разделяются запятыми и заключены в угловые скобки.
Нулевые типы	Нулевые типы — это такие типы, которые могут принимать любое значение заданного типа или значение `null`. Для объявления переменной нулевого типа может использоваться синтаксис `Nullable<T>` или `T?`.
Операция `??`	Операция объединения с `null` возвращает значение первого операнда, если он не равен `null`, или значение второго операнда в противном случае.
Обобщенные коллекции	Обобщенные коллекции исключительно полезны, поскольку поддерживают строгую типизацию. Доступны для использования типы `List<T>`, `Collection<T>` и `Dictionary<K, V>`, а также ряд других. Они также предоставляют обобщенные интерфейсы. Для сортировки и поиска применяются интерфейсы `IComparer<T>` и `IComparable<T>`.
Определение обобщенных классов	Обобщенный тип определяется во многом подобно любому другому типу, но с добавлением параметров обобщенных типов в виде заключенного в угловые скобки списка имен типов, разделенных запятыми. Параметры обобщенных типов могут использоваться в коде везде, где указываются имена типов, например, в параметрах и возвращаемых значениях методов.

Тема	Основные концепции
Ограничение параметров обобщенных типов	Чтобы более эффективно использовать параметры обобщенных типов в коде обобщенного типа, можно ограничить типы, которые допускается передавать для использования в обобщенном типе. Параметры типов можно ограничивать по базовому классу, поддерживаемому интерфейсу, принадлежности к типу значения или ссылочному типу и наличию поддержки конструкторов без параметров. Без таких ограничений для создания экземпляра переменной обобщенного типа должно использоваться ключевое слово `default`.
Другие обобщенные типы	Подобно классам, можно определять обобщенные интерфейсы, делегаты и методы.
Вариантность	Вариантность — это концепция, подобная полиморфизму, но применяется к параметрам типов. Она позволяет указывать один обобщенный тип на месте другого, при этом данные обобщенные типы могут варьироваться только в используемых параметрах обобщенных типов. Ковариантность делает возможным преобразование между двумя типами, при условии, что целевой тип имеет параметр типа, являющийся базовым классом для параметра типа в исходном типе. Контравариантность позволяет преобразование в обратном направлении. Ковариантные параметры типов определяются с помощью параметров `out` и могут использоваться только как типы возврата и типы `get`-блоков свойств. Контравариантные параметры типов определяются с помощью параметров `in` и могут использоваться только в качестве параметров методов.

13

Дополнительные приемы объектно-ориентированного программирования

В ЭТОЙ ГЛАВЕ...

- Операция : :
- Квалификатор глобального пространства имен
- Создание специальных исключений
- Использование событий
- Использование анонимных методов

В этой главе продолжается изучение языка C# и рассматриваются дополнительные приемы, которые больше никуда особо не вписывались. Это вовсе не означает их бесполезность, просто они не подпадали ни под одну из тем, которые рассматривались до сих пор.

Кроме того, в этой главе будут внесены окончательные изменения в проект `CardLib`, разрабатываемый на протяжении нескольких последних глав, и также будет показано, как его можно использовать для создания карточной игры.

Операция :: и квалификатор глобального пространства имен

Операция `::` предоставляет альтернативный способ для получения доступа к типам в пространствах имен. Это может быть необходимо, когда для пространства имен используется псевдоним и возникает неоднозначность между этим псевдонимом и фактической иерархией пространств имен. В таком случае иерархия пространств имен получает преимущество над псевдонимом. Пусть имеется следующий код:

```
using MyNamespaceAlias = MyRootNamespace.MyNestedNamespace;

namespace MyRootNamespace
{
  namespace MyNamespaceAlias
  {
    public class MyClass
    {
    }
  }

  namespace MyNestedNamespace
  {
    public class MyClass
    {
    }
  }
}
```

В коде `MyRootNamespace` для ссылки на класс мог бы применяться такой синтаксис:

```
MyNamespaceAlias.MyClass
```

Классом, на который ссылается данный код, является `MyRootNamespace.MyNamespaceAlias.MyClass`, а не `MyRootNamespace.MyNestedNamespace.MyClass`. Это значит, что пространство имен `MyRootNamespace.MyNamespaceAlias` скрыло псевдоним, определенный в операторе `using`, который ссылается на `MyRootNamespace.MyNestedNamespace`. Теперь для получения доступа к пространству имен `MyRootNamespace.MyNestedNamespace` и содержащемуся внутри него классу должен использоваться другой синтаксис:

```
MyNestedNamespace.MyClass
```

В качестве альтернативы можно применить операцию `::`, как показано ниже:

```
MyNamespaceAlias::MyClass
```

Это заставит компилятор использовать псевдоним, определенный в операторе `using`, и тогда код станет ссылаться на `MyRootNamespace.MyNestedNamespace.MyClass`.

Вместе с операцией `::` можно также применять ключевое слово `global`, что, по сути, превращает его в псевдоним, ссылающийся на корневое пространство имен наивысшего уровня. Это может быть удобно для дополнительного прояснения того, какое пространство имен имеется в виду:

```
global::System.Collections.Generic.List<int>
```

Здесь конечным классом является именно тот, который и ожидается, т.е. обобщенный класс коллекции `List<T>`, а не класс, определенный в следующем коде:

```
namespace MyRootNamespace
{
  namespace System
  {
    namespace Collections
    {
      namespace Generic
      {
        class List<T>
        {
        }
      }
    }
  }
}
```

Разумеется, следует избегать назначения собственным пространствам имен идентификаторов, которые уже есть у пространств имен, поставляемых в составе .NET. Однако в больших проектах такая проблема все равно может возникать, особенно при написании проектов многочисленной командой разработчиков. В таком случае применение операции `::` и ключевого слова `global` может оказаться единственным способом для получения доступа к желаемым типам.

Специальные исключения

В главе 7 рассказывалось об исключениях и объяснялось, как использовать блоки `try...catch...finally` для взаимодействия с ними. Там же были описаны некоторые стандартные исключения .NET, в том числе базовый класс для всех исключений — `System.Exception`. Иногда вместо работы со стандартными исключениями удобно унаследовать от этого базового класса собственные классы исключений и применять их в своих приложениях. Это позволяет передавать более специфичную информацию отвечающему за перехват исключений коду и организовать специализированную обработку исключений. Например, в класс исключения можно добавить новое свойство, которое позволит получать доступ к какой-то базовой информации. Это даст получателю исключения возможность вносить необходимые изменения либо просто извлекать дополнительные сведения о причинах возникновения исключения.

После определения класс исключения можно добавить в список исключений, распознаваемых VS. Для этого выберите в меню Debug (Отладка) пункт Exceptions (Исключения), в открывшемся после этого окне щелкните на кнопке Add (Добавить) и определите его поведение так, как было показано в главе 7.

В пространстве имен `System` существуют два фундаментальных класса исключений `ApplicationException` и `SystemException`, унаследованные от `Exception`. Класс `SystemException` используется в качестве базового для предопределенных исключений .NET Framework. Класс `ApplicationException` был предусмотрен специально для того, чтобы разработчики могли наследовать от него собственные классы исключений. Однако теперь наследовать свои исключения от этого класса не рекомендуется, а взамен следует использовать класс `Exception`. По этой причине класс `ApplicationException` со временем, скорее всего, будет объявлен устаревшим в .NET Framework.

Добавление специальных исключений в CardLib

С особенностями использования специальных исключений лучше всего знакомиться на примере обновления нашего проекта CardLib. В настоящее время метод Deck.GetCard() в этом проекте предусматривает генерацию стандартного исключения .NET при попытке получить доступ к карте с индексом меньше 0 и больше 51. Давайте изменим его так, чтобы он выдавал не стандартное, а специальное исключение.

Для этого сначала создайте новый проект библиотеки классов по имени Ch13CardLib и скопируйте в него классы из проекта Ch12CardLib, заменив в них, где требуется, пространство имен Ch12CardLib на Ch13CardLib. Чтобы определить собственное исключение, создайте следующий новый класс в файле CardOutOfRangeException.cs, добавив его в проект Ch13CardLib за счет выбора в меню Project (Проект) пункта Add Class (Добавить класс).

```
public class CardOutOfRangeException : Exception
{
  private Cards deckContents;

  public Cards DeckContents
  {
    get
    {
      return deckContents;
    }
  }
  public CardOutOfRangeException(Cards sourceDeckContents) :
      base("There are only 52 cards in the deck.")
          // В колоде есть только 52 карты
  {
    deckContents = sourceDeckContents;
  }
}
```

*Фрагмент кода **Ch13CardLib\CardOutOfRangeException.cs***

Конструктор этого класса требует экземпляра класса Cards. Это позволяет получить доступ к объекту Cards через свойство DeckContents и предоставить подходящее сообщение об ошибке конструктору базового класса Exception, чтобы оно было доступно через его свойство Message.

Теперь добавьте в Deck.cs код для генерации этого специального исключения (заменив им прежний код, который обеспечивал выдачу стандартного исключения):

```
public Card GetCard(int cardNum)
{
  if (cardNum >= 0 && cardNum <= 51)
    return cards[cardNum];
  else
    throw new CardOutOfRangeException(cards.Clone() as Cards);
}
```

*Фрагмент кода **Ch13CardLib\Deck.cs***

Свойство DeckContents инициализируется за счет глубокого копирования текущего содержимого объекта Deck в виде объекта Cards. Это значит, что будет видно содержимое на момент генерации исключения, и эта информация не "потеряется" при последующем изменении содержимого Deck.

Для целей тестирования можно воспользоваться следующим клиентским кодом (доступен в проекте Ch13CardClient):

```
Deck deck1 = new Deck();
try
{
  Card myCard = deck1.GetCard(60);
}
catch (CardOutOfRangeException e)
{
  Console.WriteLine(e.Message);
  Console.WriteLine(e.DeckContents[0]);
}
Console.ReadKey();
```

Фрагмент кода ***Ch13CardClient\Program.cs***

Выполнение этого кода приведет к получению вывода, показанного на рис. 13.1.

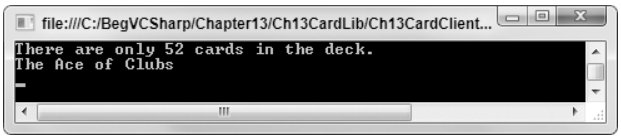


Рис. 13.1. *Генерация специального исключения в проекте* `CardLib`

Здесь код, отвечающий за перехват исключения, вывел на экран значение свойства `Message` объекта исключения. Кроме того, с помощью свойства `DeckContents` извлекается и выводится информация о первой карте в объекте `Cards`; это нужно просто для подтверждения, что доступ к коллекции `Cards` через объект специального исключения является возможным.

События

В этом разделе рассматриваются *события* (event), которые представляют собой один из наиболее часто применяемых в .NET приемов ООП. Сначала, как обычно, будут приведены базовые сведения о событиях, т.е., что они вообще собой представляют. Затем некоторые простые события демонстрируются в действии. После этого будет показано, как создавать и использовать собственные события.

В конце главы завершается разработка проекта `CardLib` за счет добавления к нему событий, а также создается несложное приложение для игры в карты, использующее эту библиотеку классов.

Что собой представляет событие

События похожи на исключения тем, что они тоже *генерируются* (выдаются) объектами, и для них можно предоставить код реакции. Однако с событиями связано несколько важных отличий, наиболее важным из которых является отсутствие для их обработки эквивалентной структуры `try...catch`. Вместо этого на события необходимо *подписываться* (subscribe). Подписание на событие означает предоставление кода, который должен выполняться при генерации данного события, в форме *обработчика событий* (event handler).

Таких обработчиков у каждого события может быть много, и все они будут вызываться в случае его генерации. Обработчики могут быть частью того же класса, в котором данное событие генерируется, но вероятнее всего они будут частью других классов.

Сами обработчики событий — это просто методы. Единственное связанное с ними ограничение состоит в том, что их возвращаемый тип и параметры должны обязательно соответствовать тем, которых требует событие. Это ограничение является частью определения события и указывается с помощью *делегата*.

Тот факт, что делегаты применяются в событиях, как раз и делает их столь полезными. Именно поэтому им было уделено внимание в главе 6, куда можно при необходимости обратиться.

Базовая последовательность обработки любого события выглядит следующим образом. Сначала в приложении создается объект, который может генерировать событие. Например, возьмем приложение для мгновенного обмена сообщениями, в котором создается объект, представляющий соединение с удаленным пользователем. Этот объект соединения может генерировать событие, когда по соединению поступает сообщение от удаленного пользователя (рис. 13.2).

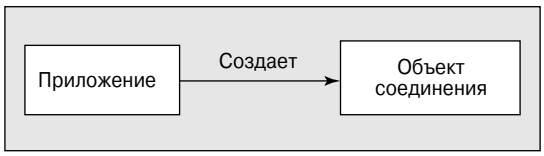


Рис. 13.2. *Объект, генерирующий события*

Далее в приложении производится подписка на созданное событие. В приложении для мгновенного обмена сообщениями это можно сделать за счет определения метода, подходящего для использования с указанным в событии типом делегата, и передачи ссылки на этот метод событию. Обработчик событий может быть методом какого-то другого объекта, например, объекта, который представляет устройство отображения для вывода мгновенных сообщений при их поступлении (рис. 13.3).

Когда событие сгенерировано, подписчик об этом уведомляется. При поступлении мгновенного сообщения через объект соединения вызывается метод обработчика событий на объекте, представляющем устройство отображения. Поскольку используется стандартный метод, объект, генерирующий событие, может передавать любую значимую информацию через параметры и тем самым делать события весьма разнообразными. В рассматриваемом примере приложения одним из параметров может быть текст мгновенного сообщения, который обработчик событий передает объекту, представляющему устройство отображения (рис. 13.4).

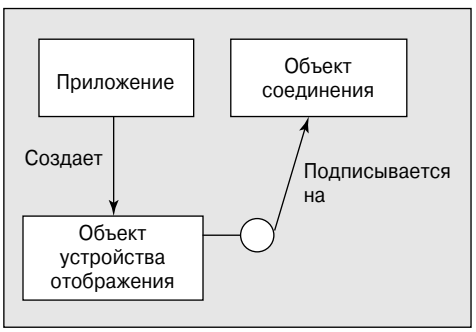


Рис. 13.3. *Подписка на событие*

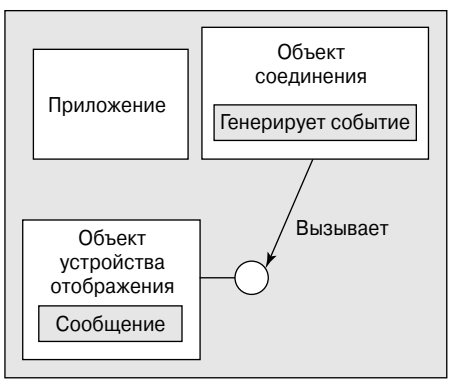


Рис. 13.4. *Генерация события*

Обработка событий

Как упоминалось ранее, для обработки события на него необходимо подписаться за счет предоставления метода-обработчика с таким же возвращаемым типом и параметрами, как у делегата, который указан для использования с этим событием. В следующем практическом занятии создается простой объект таймера для генерации событий и соответствующий метод-обработчик.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Обработка событий**

1. Создайте новое консольное приложение по имени `Ch13Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter13`.
2. Измените код в его файле `Program.cs` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Timers;

namespace Ch13Ex01
{
  class Program
  {
    static int counter = 0;
    static string displayString =
                      "This string will appear one letter at a time. ";
                  // Строка, которая будет отображаться по одной букве за раз
    static void Main(string[] args)
    {
      Timer myTimer = new Timer(100);
      myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
      myTimer.Start();
      Console.ReadKey();
    }
    static void WriteChar(object source, ElapsedEventArgs e)
    {
      Console.Write(displayString[counter++ % displayString.Length]);
    }
  }
}
```

Фрагмент кода ***Ch13Ex01\Program.cs***

3. Запустите приложение (после запуска нажатие любой клавиши приводит к его завершению). Через некоторое время на экране появится результат, показанный на рис. 13.5.

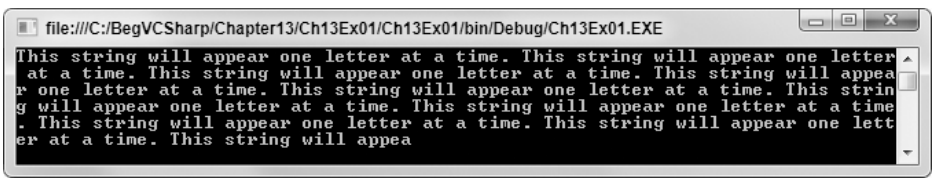


Рис. 13.5. *Приложение* `Ch13Ex01` *в действии*

Описание работы

Объект, который используется для генерации событий, представляет собой экземпляр класса `System.Timers.Timer`. Он инициализируется с указанием временного интервала (в миллисекундах). После его запуска с помощью метода `Start()` генерируется поток событий, которые разделяются промежутком времени, соответствующим указанному интервалу. В `Main()` объект `Timer` инициализируется с интервалом 100 миллисекунд, поэтому он будет генерировать события 10 раз в секунду:

```
static void Main(string[] args)
{
  Timer myTimer = new Timer(100);
```

Объект `Timer` оснащен событием по имени `Elapsed`, и обработчик этого события должен иметь такой же возвращаемый тип и параметры, как у делегата `System.Timers.ElapsedEventHandler` — одного из стандартных делегатов, поставляемых в .NET Framework. Возвращаемый тип и параметры этого делегата выглядят следующим образом:

```
void <ИмяМетода>(object source, ElapsedEventArgs e);
```

Объект `Timer` в первом параметре передает ссылку на самого себя, а во втором — экземпляр объекта `ElapsedEventArgs`. Пока на эти параметры можно не обращать внимания, т.к. они будут более подробно рассматриваться чуть позже.

В коде присутствует подходящий метод:

```
static void WriteChar(object source, ElapsedEventArgs e)
{
  Console.Write(displayString[counter++ % displayString.Length]);
}
```

В этом методе используются два статических поля из `Program` — `counter` и `displayString` — для отображения одиночного символа. При каждом вызове этого метода отображается очередной символ.

Следующим шагом является подключение этого метода-обработчика к событию, т.е. подписка на него. Для этого применяется операция +=, с помощью которой обработчик добавляется к событию в виде нового экземпляра делегата, инициализируемого методом обработчика событий `WriteChar`:

```
static void Main(string[] args)
{
  Timer myTimer = new Timer(100);
  myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

Эта команда (в которой используется немного странно выглядящий синтаксис, предназначенный специально для делегатов) обеспечивает добавление данного обработчика в список, который будет вызываться при генерации события `Elapsed`. В этот список можно добавлять сколько угодно обработчиков; главное, чтобы все они отвечали требуемым критериям. При генерации события каждый обработчик будет вызываться по очереди.

После этого остается лишь запустить таймер в `Main()`:

```
myTimer.Start();
```

Поскольку работа приложения не должна завершаться до того, как будут обработаны хоть какие-то события, метод `Main()` должен быть переведен в режим ожидания. Простейшим способом является запрос ввода пользователя, поскольку такая команда не позволит завершиться процессу обработки до тех пор, пока пользователь не нажмет какую-нибудь клавишу:

```
Console.ReadKey();
```

Хотя в `Main()` обработка на этом этапе фактически прекращена, в объекте `Timer` она продолжается. При генерации событий в нем вызывается метод `WriteChar()`, который работает вместе с оператором `Console.ReadLine()`.

Обратите внимание, что синтаксис для добавления обработчика событий можно немного упростить, применив концепцию группы методов, о которой рассказывалось в предыдущей главе:

```
myTimer.Elapsed += WriteChar;
```

Конечный результат будет выглядеть точно так же, но зато явно указывать тип делегата больше не требуется, поскольку компилятор будет вычислять его сам из того контекста, в котором он используется. Однако многим программистам такой синтаксис не нравится из-за того, что он делает код менее удобным для восприятия; например, здесь невозможно сразу понять, делегат какого типа используется. Желающие могут спокойно пользоваться группами методов, но для большей ясности в этой главе типы всех применяемых делегатов будут указываться явно.

Определение событий

Теперь пришла пора посмотреть, как определять и использовать собственные события. Следующее практическое занятие посвящено реализации описанного выше приложения для мгновенного обмена сообщениями, в котором создается объект соединения `Connection`, способный генерировать события, и объект устройства отображения `Display`, отвечающий за их обработку.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Определение событий**

1. Создайте новое консольное приложение по имени `Ch13Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter13`.
2. Добавьте в него новый класс `Connection` и измените код в файле `Connection.cs` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Timers;
namespace Ch13Ex02
{
  public delegate void MessageHandler(string messageText);
  public class Connection
  {
    public event MessageHandler MessageArrived;
    private Timer pollTimer;
    public Connection()
    {
       pollTimer = new Timer(100);
       pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
    }
    public void Connect()
    {
      pollTimer.Start();
    }
    public void Disconnect()
    {
       pollTimer.Stop();
    }
```

```
    private static Random random = new Random();

    private void CheckForMessage(object source, ElapsedEventArgs e)
     {
        Console.WriteLine("Checking for new messages.");
                    // Проверка на предмет поступления новых сообщений
        if ((random.Next(9) == 0) && (MessageArrived != null))
     {
        MessageArrived("Hello Mum!");
                   // Новое сообщение
     }
     }
   }
}
```

Фрагмент кода ***Ch13Ex02\Connection.cs***

3. Добавьте новый класс `Display` и модифицируйте его код в файле `Display.cs`, как показано ниже:

```
namespace Ch13Ex02
{
   public class Display
   {
      public void DisplayMessage(string message)
      {
        Console.WriteLine("Message arrived: {0}", message);
                        // Поступило новое сообщение
      }
   }
}
```

Фрагмент кода ***Ch13Ex02\Display.cs***

4. Измените код в файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  Connection myConnection = new Connection();
  Display myDisplay = new Display();
  myConnection.MessageArrived +=
    new MessageHandler (myDisplay.DisplayMessage);
  myConnection.Connect();
  Console.ReadKey();
}
```

Фрагмент кода ***Ch13Ex02\Program.cs***

5. Запустите приложение. На рис. 13.6 показан результат, который должен получиться.

Описание работы

Большую часть работы в этом приложении выполняет класс `Connection`. Его экземпляры используют объект `Timer`, похожий на применяемый в первом примере настоящей главы. Этот объект инициализируется в конструкторе класса, а доступ к его состоянию (включен или отключен) производится с помощью методов `Connect()` и `Disconnect()`.

```
public class Connection
{
  private Timer pollTimer;

  public Connection()
  {
    pollTimer = new Timer(100);
```

```
    pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
  }
  public void Connect()
  {
    pollTimer.Start();
  }
  public void Disconnect()
  {
    pollTimer.Stop();
  }
  ...
}
```

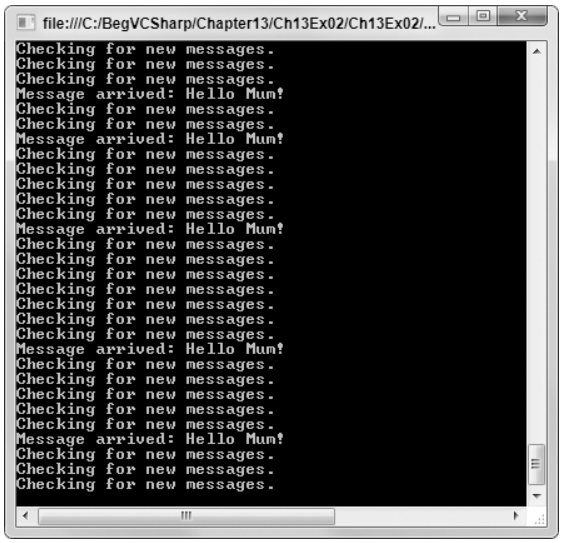


Рис. 13.6. *Приложение* `Ch13Ex02` *в действии*

В конструкторе также производится регистрация обработчика для события `Elapsed`, как это делалось в первом примере. Метод этого обработчика, `CheckForMessage()`, генерирует событие в среднем один раз на каждые 10 его вызовов. Его код будет поясняться немного позже, а сначала рассмотрим определение самого события.

Перед определением события должен быть определен тип делегата для использования с событием, т.е. тип делегата, возвращаемый тип и параметры которого должны совпадать с таковыми метода-обработчика. Это делается с помощью стандартного синтаксиса для делегатов, согласно которому делегат делается общедоступным внутри пространства имен `Ch13Ex02`, что позволит получать к нему доступ из внешнего кода:

```
namespace Ch13Ex02
{
  public delegate void MessageHandler(string messageText);
```

Этот тип делегата, называющийся здесь `MessageHandler`, представляет собой метод `void`, имеющий единственный параметр `string`. Этот параметр можно использовать для передачи объекту `Display` мгновенного сообщения, полученного объектом `Connection`. После определения делегата (или нахождения подходящего существующего делегата) можно переходить к определению самого события, в виде члена класса `Connection`:

```
public class Connection
{
  public event MessageHandler MessageArrived;
```

Событию назначается имя (MessageArrived) и затем оно объявляется с применением ключевого слова event и указанием используемого с ним типа делегата (ранее определенный тип MessageHandler). Сразу после объявления событие можно генерировать, просто обращаясь к нему по имени так, как если бы оно являлось методом с возвращаемым типом и параметрами, заданными в делегате. Например, для вызова определенного выше события можно применить такой код:

```
MessageArrived("This is a message.");
```

Если бы делегат был определен без параметров, тогда подошел бы следующий код:

```
MessageArrived();
```

В качестве альтернативы можно определить большее число параметров, и тогда нужно будет писать больше кода для генерации события. Метод CheckForMessage() выглядит следующим образом:

```
private static Random random = new Random();

private void CheckForMessage(object source, ElapsedEventArgs e)
{
  Console.WriteLine("Checking for new messages.");
                  // Проверка на предмет поступления новых сообщений
  if ((random.Next(9) == 0) && (MessageArrived != null))
  {
    MessageArrived("Hello Mum!");
                 // Новое сообщение
  }
}
```

В методе используется экземпляр класса Random для генерации случайного числа в диапазоне от 0 до 9, а событие генерируется, если полученное случайное число равно 0, что должно происходить в 10% случаев. Это имитирует опрос подключения на предмет поступления нового сообщения, чего не случается при каждой проверке. Для отделения таймера от экземпляра Connection применяется приватный статический экземпляр класса Random.

Обратите внимание на предоставление дополнительной логики. Событие генерируется, только если выражение MessageArrived != null дает true. Это выражение, в котором снова используется несколько необычный синтаксис делегатов, запрашивает, есть ли у события какие-то подписчики. Если таковых нет, MessageArrived равно null, и событие генерировать не имеет смысла.

Класс, который будет подписываться на событие, называется Display и содержит единственный метод по имени DisplayMessage(), определение которого приведено ниже:

```
public class Display
{
  public void DisplayMessage(string message)
  {
    Console.WriteLine("Message arrived: {0}", message);
                   // Поступило новое сообщение
  }
}
```

Этот метод соответствует типу делегата (и является общедоступным, что представляет собой требование для всех обработчиков событий, которые используются в классах, отличных от класса, генерирующего событие), поэтому может применяться для реагирования на событие MessageArrived.

Теперь осталось только сделать так, чтобы код в Main() инициализировал экземпляры классов Connection и Display, подключал их и запускал весь процесс. Необходимый для этого код похож на код из первого примера:

```
static void Main(string[] args)
{
  Connection myConnection = new Connection();
  Display myDisplay = new Display();
  myConnection.MessageArrived +=
         new MessageHandler(myDisplay.DisplayMessage);
  myConnection.Connect();
  Console.ReadKey();
}
```

Здесь также вызывается метод `Console.ReadKey()` для приостановки обработки в `Main()` после запуска процесса с помощью метода `Connect()` объекта `Connection`.

Универсальные обработчики событий

Делегат, показанный ранее для события `Timer.Elapsed`, содержал два параметра, которые часто встречаются в обработчиках событий:

- `object source` — ссылка на объект, который сгенерировал событие;
- `ElapsedEventArgs e` — параметры, отправленные событием.

Причина использования параметра `object` в этом и многих других событиях связана с тем, что часто один обработчик событий необходимо применять для нескольких идентичных событий, генерируемых разными объектами, и при этом знать, какой объект сгенерировал данное событие.

Чтобы лучше объяснить и проиллюстрировать это, в следующем практическом занятии будет немного расширен предыдущий пример.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Применение универсального обработчика событий**

1. Создайте новое консольное приложение по имени `Ch13Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter13`.
2. Скопируйте код из файлов `Program.cs`, `Connection.cs` и `Display.cs` проекта `Ch13Ex02` и замените названия пространств имен в каждом из них с `Ch13Ex02` на `Ch13Ex03`.
3. Добавьте новый класс `MessageArrivedEventArgs` и измените код в представляющем его файле `MessageArrivedEventArgs.cs`, как показано ниже:

```
namespace Ch13Ex03
{
  public class MessageArrivedEventArgs : EventArgs
  {
    private string message;
    public string Message
    {
      get
      {
        return message;
      }
    }

    public MessageArrivedEventArgs()
    {
      message = "No message sent.";
              // Новых сообщений нет
    }
```

```
    public MessageArrivedEventArgs(string newMessage)
    {
      message = newMessage;
    }
  }
}
```

Фрагмент кода ***Ch13Ex03\MessageArrivedEventArgs.cs***

4. Модифицируйте код в файле `Connection.cs` следующим образом:

```
namespace Ch13Ex03
{
  public delegate void MessageHandler(Connection source,
                                      MessageArrivedEventArgs e);
  public class Connection
  {
    public event MessageHandler MessageArrived;
    public string Name { get; set; }
    ...
    private void CheckForMessage(object source, EventArgs e)
    {
      Console.WriteLine("Checking for new messages.");
                        // Проверка на предмет поступления новых сообщений
      if ((random.Next(9) == 0) && (MessageArrived != null))
      {
        MessageArrived(this, new MessageArrivedEventArgs("Hello Mum!"));
      }
    }
    ...
  }
}
```

Фрагмент кода ***Ch13Ex03\Connection.cs***

5. Измените содержимое файла `Display.cs` так, как показано ниже:

```
public void DisplayMessage(Connection source, MessageArrivedEventArgs e)
{
  Console.WriteLine("Message arrived from: {0}", source.Name);
                  // Поступило новое сообщение
  Console.WriteLine("Message Text: {0}", e.Message);
                  // Текст сообщения
}
```

Фрагмент кода ***Ch13Ex03\Display.cs***

6. И, наконец, модифицируйте код в файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  Connection myConnection1 = new Connection();
  myConnection1.Name = "First connection.";     // Первое подключение
  Connection myConnection2 = new Connection();
  myConnection2.Name = "Second connection.";    // Второе подключение
  Display myDisplay = new Display();
  myConnection1.MessageArrived +=
            new MessageHandler(myDisplay.DisplayMessage);
  myConnection2.MessageArrived +=
            new MessageHandler(myDisplay.DisplayMessage);
  myConnection1.Connect();
  myConnection2.Connect();
  Console.ReadKey();
}
```

Фрагмент кода ***Ch13Ex03\Program.cs***

7. Запустите приложение. На рис. 13.7 показан результат, который должен получиться.

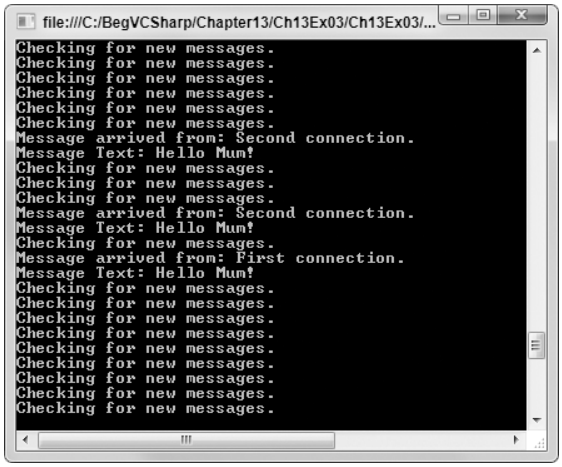


Рис. 13.7. *Приложение* `Ch13Ex03` *в действии*

Описание работы

Передавая обработчику событий ссылку на генерирующий событие объект в виде одного из параметров, можно настраивать ответ этого обработчика для отдельных объектов. Эта ссылка предоставляет доступ к исходному объекту, включая его свойства.

За счет передачи параметров, содержащихся в классе, который наследуется от `System.EventArgs` (как это делает `ElapsedEventArgs`), можно предоставлять любую необходимую информацию (такую как параметр `Message` в классе `MessageArrivedEventArgs`).

Вдобавок эти параметры будут извлекать пользу из полиморфизма. Обработчик для события `MessageArrived` можно было бы определить так, как показано ниже:

```
public void DisplayMessage(object source, EventArgs e)
{
  Console.WriteLine("Message arrived from: {0}",
                    //Поступило новое сообщение от: {0}
          ((Connection)source).Name);
  Console.WriteLine("Message Text: {0}",
                   // Текст этого сообщения выглядит так: {0}
          ((MessageArrivedEventArgs)e).Message);
}
```

В таком случае определение делегата в `Connection.cs` можно было бы изменить следующим образом:

```
public delegate void MessageHandler(object source, EventArgs e);
```

Приложение продолжило бы работать точно так же, но его метод `DisplayMessage()` стал бы более универсальным (во всяком случае, теоретически; на практике для достижения более высокого качества потребовалось бы написать больше кода реализации). Тот же самый обработчик смог бы работать с другими событиями, такими как `Timer.Elapsed`. Правда, пришлось бы еще немного изменить внутренние детали обработчика, чтобы параметры, отправляемые при генерации данного события, обрабатывались надлежащим образом (приведение их к типу объектов `Connection` и `MessageArrivedEventArgs` показанным способом привело бы к генерации исключения; потребовалось бы использовать операцию `as` и выполнять проверку на предмет `null`).

Тип EventHandler и обобщенный тип EventHandler<T>

В большинстве случаев вы будете следовать схеме, описанной в предыдущем разделе, и использовать обработчики событий с возвращаемым типом `void` и двумя параметрами. Первый параметр относится к типу `object` и представляет источник события, а второй — к типу, унаследованному от `System.EventArgs`, и содержит любые аргументы события. Поскольку схема является очень типичной, в .NET для упрощения определения событий предлагается два специальных типа делегата: `EventHandler` и `EventHandler<T>`. Оба они используют стандартную схему для обработчиков событий. Обобщенная версия позволяет указывать необходимый тип аргумента события.

Следовательно, вместо определения собственного типа делегата `MessageHandler`, как было сделано в предыдущем примере, событие `MessageArrived` можно определить следующим образом:

```
public class Connection
{
  public event EventHandler MessageArrived;
  ...
}
```

или даже таким образом:

```
public class Connection
{
  public event EventHandler<MessageArrivedEventArgs> MessageArrived;
  ...
}
```

Это очевидно удобно, поскольку существенно упрощает код.

Возвращаемые значения и обработчики событий

Все показанные до сих пор обработчики событий имели возвращаемый тип `void`. Предоставлять другой возвращаемый тип для события в принципе допускается, но это чревато возникновением проблем, поскольку отдельно взятое событие может приводить к вызову множества обработчиков. Если все они возвращают значение, в результате может быть неясно, какое конкретно значение было в действительности возвращено.

Система справляется с этой проблемой, разрешая получать доступ только к тому значению, которое было возвращено обработчиком событий последним. Это всегда будет значение, возвращаемое обработчиком, который подписался на данное событие последним. Хотя в некоторых ситуациях эта функциональность подходит, рекомендуется применять обработчики событий с возвращаемым типом `void` и избегать использования в них параметров `out` (которые тоже могут приводить к такой же неоднозначности, но только в отношении источника возвращаемого параметром значения).

Анонимные методы

Вместо того чтобы определять методы для обработки событий, можно воспользоваться *анонимными методами*. Анонимным называется метод, который в действительности не существует в виде метода в традиционном смысле, т.е. он не является методом какого-то класса. Вместо этого анонимный метод создается исключительно для применения в качестве целевого метода для делегата.

Ниже показан синтаксис, необходимый для создания анонимного метода:

```
delegate(параметры)
{
  // Код анонимного метода.
};
```

На месте *параметры* указывается список параметров, соответствующих параметрам делегата, экземпляр которого создается, в том виде, в котором они должны использоваться в коде анонимного метода:

```
delegate(Connection source, MessageArrivedEventArgs e)
{
  // Код анонимного метода, соответствующего событию MessageHandler в Ch13Ex03.
};
```

Например, с помощью следующего кода можно полностью обойти метод `Display.DisplayMessage()` в `Ch13Ex03`:

```
myConnection1.MessageArrived +=
delegate(Connection source, MessageArrivedEventArgs e)
{
  Console.WriteLine("Message arrived from: {0}", source.Name);
                  // Поступило новое сообщение
  Console.WriteLine("Message Text: {0}", e.Message);
                  // Текст сообщения
};
```

Интересное свойство анонимных методов состоит в том, что они, по сути, являются локальными для блока кода, в котором содержатся, и имеют доступ ко всем локальным переменным в этой области действия. В случае использования такой переменной она становится *внешней*. Внешние переменные не уничтожаются при выходе за пределы области действия, как это происходит с другими локальными переменными; вместо этого они продолжают существовать до тех пор, пока не будут уничтожены те методы, в которых они используются. Это может произойти позже, чем ожидалось, потому требует внимательного отношения. Если внешняя переменная занимает слишком большой объем памяти либо использует другие дорогостоящие ресурсы (например, ограниченные ресурсы), могут возникать проблемы с памятью или производительностью.

Расширение и использование `CardLib`

Теперь, когда было показано, как определять и использовать события, пришла пора попробовать применить полученные знания на практике — в проекте `Ch13CardLib`. В этом разделе в библиотеку будет добавлено событие по имени `LastCardDrawn` (последняя вытянутая карта), которое генерируется при получении последнего объекта `Card` в объекте `Deck` с использованием `GetCard`. Оно позволяет подписчикам автоматически перемешивать колоду, тем самым сокращая объем обработки, которую должен выполнять клиент. Делегат, определенный для этого события (`LastCardDrawnHandler`) должен поддерживать ссылку на объект `Deck`, чтобы метод `Shuffle()` мог быть доступен отовсюду, где бы ни находился обработчик. Добавьте в `Deck.cs` следующий код:

```
namespace Ch13CardLib
 {
  public delegate
    void LastCardDrawnHandler(Deck currentDeck);
```

Фрагмент кода ***Ch13CardLib\Deck.cs***

Код определения описанного события и его вызова достаточно прост:

```
public event LastCardDrawnHandler LastCardDrawn;
...
public Card GetCard(int cardNum)
{
  if (cardNum >= 0 && cardNum <= 51)
  {
```

```
    if ((cardNum == 51) && (LastCardDrawn != null))
      LastCardDrawn(this);
    return cards[cardNum];
  }
  else
    throw new CardOutOfRangeException((Cards)cards.Clone());
}
```

Это весь код, необходимый для добавления события к определению класса `Deck`.

Клиентское приложение для игры в карты, использующее библиотеку `CardLib`

После такого количества усилий, потраченных на разработку библиотеки `CardLib`, вполне уместно поработать с ней. Перед завершением этого раздела ООП на C# и .NET Framework давайте немного развлечемся и построим простейшее клиентское приложение игры в карты, в котором будут использоваться уже знакомые классы игральных карт.

Как и в предыдущих главах, сначала добавьте в решение `Ch13CardLib` новое клиентское консольное приложение, добавьте в него ссылку на проект `Ch13CardLib` и сделайте его стартовым. Назовите приложение `Ch13CardClient`.

Создайте в `Ch13CardClient` новый класс по имени `Player` внутри нового файла `Player.cs`. Этот класс будет иметь два автоматических свойства: `Name` (типа `string`) и `PlayHand` (типа `Cards`). Оба свойства должны иметь приватные блоки `get`, а свойство `PlayHand` должно также предоставлять доступ для записи к своему содержимому, позволяя изменять карты, имеющиеся на руках у игрока.

Также скройте конструктор по умолчанию, сделав его приватным, и предоставьте общедоступный конструктор не по умолчанию, принимающий в качестве параметра начальное значение для свойства `Name` экземпляров `Player`.

И, наконец, создайте метод типа `bool` по имени `HasWon()`, который возвращает `true`, если все карты на руках игрока являются одной масти (простое условие для победы, но для этого примера достаточно).

Ниже показан код из файла `Player.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;

namespace Ch13CardClient
{
  public class Player
  {
    public string Name { get; private set; }
    public Cards PlayHand { get; private set; }

    private Player()
    {
    }
    public Player(string name)
    {
      Name = newName;
      Playhand = new Cards();
    }
    public bool HasWon()
    {
      bool won = true;
      Suit match = PlayHand[0].suit;
```

```
      for (int i = 1; i < PlayHand.Count; i++)
      {
        won & = PlayHand[i].suit == match;
      }
      return won;
    }
  }
}
```

Фрагмент кода ***Ch13CardClient\Player.cs***

Далее определите класс по имени `Game`, который будет отвечать за саму карточную игру. Этот класс должен быть помещен в файл `Game.cs` проекта `Ch13CardClient` и имеет четыре приватных поля.

- `playDeck` — переменная типа `Deck`, в которую должна помещаться используемая колода карт;
- `currentCard` — значение типа `int`, которое должно использоваться в качестве указателя на следующую карту в используемой колоде;
- `players` — массив объектов `Player`, представляющих игроков игры;
- `discardedCards` — коллекция `Cards`, куда должны помещаться карты, которые были отброшены игроками, но не были перетасованы и возвращены обратно в колоду.

Конструктор по умолчанию этого класса инициализирует и тасует хранящийся в `playDeck` объект `Deck`, устанавливает переменную указателя `currentCard` в `0` (первая карта в `playDeck`) и подключает обработчик событий по имени `Reshuffle()` к событию `playDeck.LastCardDrawn`. Обработчик просто тасует колоду, инициализирует коллекцию `discardedCards` и сбрасывает `currentCard` в `0`, что означает готовность считывать карты из новой колоды.

Класс `Game` также содержит два служебных метода: `SetPlayers()` для установки игроков для игры (в виде массива объектов `Player`) и `DealHands()` для раздачи карт на руки игрокам (по семь карт каждому). Допустимое количество игроков ограничивается диапазоном от 2 до 7, чтобы гарантировать наличие достаточного числа карт для хождения по кругу.

И, наконец, имеется метод `PlayGame()`, который содержит логику самой игры. Мы еще вернемся к этому методу позже, после рассмотрения кода в `Program.cs`. Ниже показан остаток кода из `Game.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;
namespace Ch13CardClient
{
  public class Game
  {
    private int currentCard;
    private Deck playDeck;
    private Player[] players;
    private Cards discardedCards;

    public Game()
    {
      currentCard = 0;
      playDeck = new Deck(true);
      playDeck.LastCardDrawn += new LastCardDrawnHandler(Reshuffle);
      playDeck.Shuffle();
      discardedCards = new Cards();
```

```
    }
  private void Reshuffle(Deck currentDeck)
  {
    Console.WriteLine("Discarded cards reshuffled into deck.");
                    // Отброшенные карты перетасованы и помещены в колоду
    currentDeck.Shuffle();
    discardedCards.Clear();
    currentCard = 0;
  }

  public void SetPlayers(Player[] newPlayers)
  {
    if (newPlayers.Length > 7)
     throw new ArgumentException("A maximum of 7 players may play this" +
                                 " game.");
                                // В эту игру может играть не более 7 игроков
    if (newPlayers.Length < 2)
      throw new ArgumentException("A minimum of 2 players may play this" +
                                 " game.");
                                // В эту игру может играть не менее 2 игроков
    players = newPlayers;
  }

  private void DealHands()
  {
    for (int p = 0; p < players.Length; p++)
    {
      for (int c = 0; c < 7; c++)
      {
        players[p].PlayHand.Add(playDeck.GetCard(currentCard++));
      }
    }
  }

  public int PlayGame()
  {
    // Код, который должен выполняться дальше.
  }
 }
}
```

Фрагмент кода ***Ch13CardClient\Game.cs***

В `Program.cs` содержится метод `Main()`, который инициализирует и запускает игру. Этот метод выполняет перечисленные ниже шаги.

1. Отображается вводная информация.
2. У пользователя запрашивается количество игроков (от двух до семи).
3. Должным образом настраивается массив объектов `Player`.
4. У каждого игрока запрашивается имя, которое используется для инициализации соответствующего объекта `Player` в массиве.
5. Создается объект `Game` и с помощью метода `SetPlayers()` назначаются игроки.
6. С помощью метода `PlayGame()` запускается игра.
7. Значение `int`, возвращаемое из `PlayGame()`, используется для вывода сообщения о победителе (возвращаемое значение — это индекс победившего игрока в массиве объектов `Player`).

Реализующий перечисленные выше шаги код выглядит следующим образом:

```
static void Main(string[] args)
{
  // Отображение вводной информации.
  Console.WriteLine("KarliCards: a new and exciting card game.");
                  // KarliCards: новая и увлекательная карточная игра
  Console.WriteLine("To win you must have 7 cards of the same suit in" +
                    "your hand.");
  // Для выигрыша необходимо, чтобы на руках оказалось 7 карт одной масти
  Console.WriteLine();

  // Запрос количества игроков.
  bool inputOK = false;
  int choice = -1;
  do
  {
    Console.WriteLine("How many players (2-7)?"); // Ввод количества игроков (2-7)
    string input = Console.ReadLine();
    try
    {
      // Попытка преобразовать введенные данные в допустимое число игроков.
      choice = Convert.ToInt32(input);
      if ((choice >= 2) && (choice <= 7))
        inputOK = true;
    }
    catch
    {
      // Игнорирование неудачных попыток преобразования
      // и продолжение запроса на ввод.
    }
  } while (inputOK == false);

  // Инициализация массива объектов Player.
  Player[] players = new Player[choice];

  // Получение имен игроков.
  for (int p = 0; p < players.Length; p++)
  {
    Console.WriteLine("Player {0}, enter your name:", p + 1);
                    // Ввод имени игрока
    string playerName = Console.ReadLine();
    players[p] = new Player(playerName);
  }

  // Запуск игры.
  Game newGame = new Game();
  newGame.SetPlayers(players);
  int whoWon = newGame.PlayGame();

  // Вывод сообщения о победившем игроке.
  Console.WriteLine("{0} has won the game!", players[whoWon].Name);
}
```

Фрагмент кода ***Ch13CardClient\Program.cs***

Теперь давайте рассмотрим код метода `PlayGame()`, являющегося основной частью приложения. Мы опустим описание всех его деталей, поскольку код достаточно хорошо прокомментирован. Ничего особо сложного в этом методе нет.

Игра продолжается тем, что каждый из игроков просматривает свои карты и ту, что была выложена в перевернутом виде на стол. Игроки могут либо брать эту карту, либо вытаскивать новую из колоды. После вытаскивания карты каждый игрок должен обязательно отбрасывать какую-то одну карту и либо заменять ею ту, что взял со стола, либо класть ее поверх нее (также добавляя отброшенную карту в коллекцию `discardedCards`).

При изучении данного кода обратите внимание на то, как производится манипулирование объектами `Card`. Причина определения этих объектов как ссылочных типов, а не типов значения (с использованием структуры), теперь должна быть совершенно ясной. Любой объект `Card` может находиться одновременно в нескольких местах, поскольку ссылка не него может храниться в объекте `Deck`, в полях `hand` объектов `Player`, в коллекции `discardedCards`, а также в объекте `playCard` (карта, выложенная на стол в текущий момент). Это упрощает отслеживание карт и, в частности, применяется в коде, отвечающем за вытаскивание новой карты из колоды. Карта принимается, только если она отсутствует на руках у игроков и в коллекции `discardedCards`.

Ниже показан код метода `PlayGame()`.

```
public int PlayGame()
{
  // Проводить игру, только если существуют игроки.
  if (players == null)
  return -1;

  // Первая раздача карт на руки игрокам.
  DealHands();

  // Инициализация имеющих отношение к игре переменных, включая
  // переменную, которая представляет первую карту на столе: playCard.
  bool GameWon = false;
  int currentPlayer;
  Card playCard = playDeck.GetCard(currentCard++);
  discardedCards.Add(playCard);

  // Главный цикл игры, который выполняется, пока GameWon == false.
  do
  {
    // Проход по игрокам в каждом раунде игры.
    for (currentPlayer = 0; currentPlayer < players.Length;
        currentPlayer++)
    {
    // Вывод текущего игрока, его карт и карты на столе.
    Console.WriteLine("{0}'s turn.", players[currentPlayer].Name);
    Console.WriteLine("Current hand:");
    foreach (Card card in players[currentPlayer].PlayHand)
    {
      Console.WriteLine(card);
    }
    Console.WriteLine("Card in play: {0}", playCard);
    // Приглашение игроку взять карту со стола или вытащить новую.
    bool inputOK = false;
    do
    {
      Console.WriteLine("Press T to take card in play or D to " +
                        "draw:");
                    // Нажмите T, чтобы взять разыгранную карту,
                    // или D, чтобы вытащить новую
      string input = Console.ReadLine();
      if (input.ToLower() == "t")
      {
        // Добавление карты со стола на руки игроку.
        Console.WriteLine("Drawn: {0}", playCard);
        // Удаление отброшенных карт, если возможно (если
        // колода перемешана, их там больше быть не должно).
        if (discardedCards.Contains(playCard))
        {
          discardedCards.Remove(playCard);
        }
```

```
      players[currentPlayer].PlayHand.Add(playCard);
      inputOK = true;
    }

    if (input.ToLower() == "d")
    {
      // Добавление на руки игроку новой карты из колоды.
      Card newCard;

      // Добавление карты только в том случае, если ее нет
      // на руках у игроков и в отброшенных картах.
      bool cardIsAvailable;
      do
      {
        newCard = playDeck.GetCard(currentCard++);

       // Проверка, не находится ли карта в отброшенных картах.
       cardIsAvailable = !discardedCards.Contains(newCard);
       if (cardIsAvailable)
       {
         // Просмотр карт на руках у всех игроков для выяснения,
         // не находится ли карта newCard у кого-нибудь из них.
          foreach (Player testPlayer in players)
          {
            if (testPlayer.PlayHand.Contains(newCard))
            {
              cardIsAvailable = false;
              break;
            }
          }
       }
    } while (!cardIsAvailable);
    // Добавление найденной карты на руки игроку.
    Console.WriteLine("Drawn: {0}", newCard);
    players[currentPlayer].PlayHand.Add(newCard);
    inputOK = true;
  }
} while (inputOK == false);

// Отображение новой раскладки на руках у игрока с нумерацией карт.
Console.WriteLine("New hand:");
for (int i = 0; i < players[currentPlayer].PlayHand.Count; i++)
{
  Console.WriteLine("{0}: {1}", i + 1,
                players[currentPlayer].PlayHand[i]);
}

// Приглашение игроку отбросить какую-нибудь карту.
inputOK = false;
int choice = -1;
do
{
  Console.WriteLine("Choose card to discard:");
                  // Выберите карту для отбрасывания
  string input = Console.ReadLine();
  try
{
   // Попытка преобразовать введенные
   // данные в допустимый номер карты.
   choice = Convert.ToInt32(input);
   if ((choice > 0) && (choice <= 8))
     inputOK = true;
}
```

```
 catch
 {
   // Игнорирование неудачных попыток преобразования
   // и продолжение вывода приглашения.
 }
} while (inputOK == false);

// Помещение ссылки на удаляемую карту в playCard (выкладывание карты на стол),
// затем изъятие карты из рук игрока и добавление ее в стопку отброшенных карт.
playCard = players[currentPlayer].PlayHand[choice - 1];
players[currentPlayer].PlayHand.RemoveAt(choice - 1);
discardedCards.Add(playCard);
Console.WriteLine("Discarding: {0}", playCard);   // Отбрасывание карты

// Вывод пустой строки для удобства.
Console.WriteLine();

// Проверка, выиграл ли игрок в этой игре, и если да, то выход из цикла.
GameWon = players[currentPlayer].HasWon();
if (GameWon == true)
  break;
 }
} while (GameWon == false);

// Завершение игры с указанием выигравшего игрока.
return currentPlayer;
}
```

На рис. 13.8 можно видеть приложение карточной игры в действии.

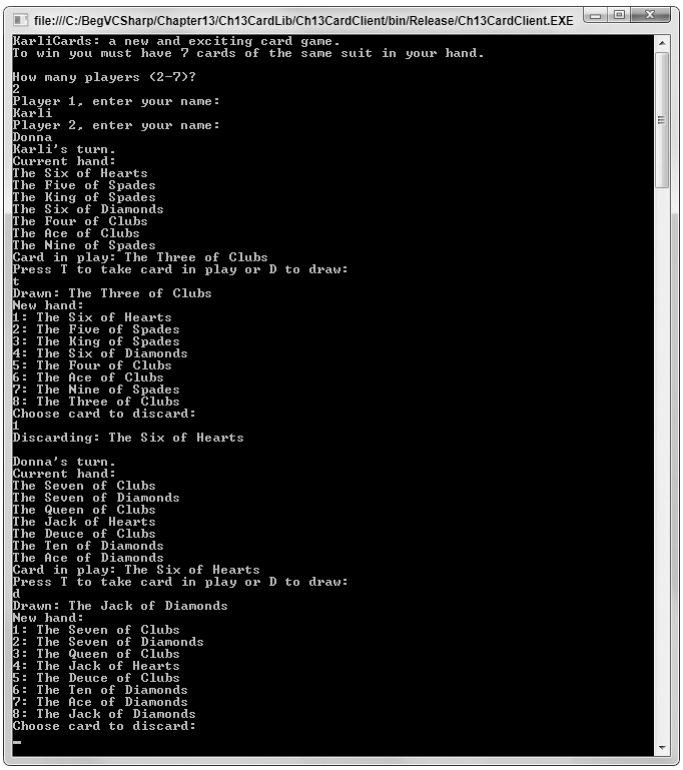


Рис. 13.8. *Карточная игра в действии*

Обязательно поиграйте в эту игру и не пожалейте времени на изучение деталей кода. Например, попробуйте разместить в методе `Reshuffle()` точку останова и сыграть в игру с участием семи игроков. В этом случае при постоянном вытаскивании и отбрасывании вытянутых карт перетасовка колоды будет происходить очень часто, т.к. при наличии семи игроков в запасе будет оставаться только три карты. Это позволит лучше изучить работу кода.

Резюме

В этой главе рассматривались некоторые расширенные приемы программирования на C#. В начале была описана квалификация пространств имен, операция `::` и ключевое слово `global`, которые обеспечивают ссылку на нужные типы. Затем было показано, как реализовать собственные объекты исключений и передавать обработчикам исключений более детальную информацию. После этого специальное исключение было добавлено в код библиотеки `CardLib`, разрабатываемой на протяжении нескольких последних глав.

И, наконец, подробно рассматривалась такая важная тема, как события и способы их обработки. Требуемый для этого код, хоть поначалу и выглядит замысловатым, в действительности достаточно прост, а обработчики событий будут повсеместно применяться далее в книге. Было показано несколько наглядных примеров событий и их обработки, а также внесено соответствующее изменение в библиотеку `CardLib`. Кроме того, было продемонстрировано использование этой библиотеки при создании простого приложения карточной игры, в котором применялись практически все приемы, рассмотренные до сих пор.

На этом рассмотрение приемов ООП на C# и всех базовых возможностей языка завершено. В следующей главе речь пойдет о новых средствах C#, которые появились в его версиях 3 и 4.

Упражнения

1. Напишите обработчик событий, который использует универсальный синтаксис `(object sender, EventArgs e)` и принимает либо событие `Timer.Elapsed`, либо событие `Connection.MessageArrived` из кода, приведенного ранее в главе. Этот обработчик должен выводить строку, указывающую тип полученного события, а также значение свойства `Message` параметра `MessageArrivedEventArgs` или свойства `SignalTime` параметра `ElapsedEventArgs` в зависимости от произошедшего события.
2. Модифицируйте пример приложения карточной игры, добавив проверку более интересного условия выигрыша, как в популярной карточной игре “рамми”. Выигравшим при этом считается игрок, у которого на руках оказывается два “набора” карт — один из трех карт и второй из четырех карт. Под набором понимается либо последовательность карт одной масти (например, 3 червей, 4 червей, 5 червей, 6 червей), либо несколько карт одного достоинства (например, 2 червей, 2 пик, 2 бубен).
   Решения упражнений приведены в приложении A.

Что вы узнали в этой главе

Тема	Основные концепции
Квалификация пространств имен	Чтобы избежать неоднозначность при квалификации пространств имен, можно применять операцию `::`, которая заставляет компилятор использовать созданные в коде псевдонимы. С помощью `global` указывается псевдоним пространства имен наивысшего уровня.
Собственные исключения	Собственные классы исключений создаются за счет наследования от корневого класса `Exception`. Это позволяет получить больший контроль над перехватом специфических исключений и настраивать данные, передаваемые вместе с исключением.
Обработка событий	Многие классы поддерживают события, которые генерируются при достижении определенных условий в их коде. Для таких событий можно создавать обработчики, которые позволяют выполнять код в ответ на возникновение события. Эти двунаправленные коммуникации представляют собой великолепный механизм для построения отзывчивого кода и существенно упрощают решение задач, связанных с опросом состояния объектов.
Определение событий	Определение собственных типов событий предусматривает создание именованного события и типов делегатов для любых его обработчиков. Для получения универсальных обработчиков событий можно использовать стандартный тип делегата, который не имеет возвращаемого типа и получает собственные аргументы события, унаследованные от `System.EventArgs`. Для упрощения кода определения событий можно также использовать типы делегатов `EventHandler` и `EventHandler<T>`.
Анонимные методы	Часто для повышения читабельности кода можно использовать анонимный метод, а не полный метод обработчика события. Это означает, что код, выполняемый при возникновении события, определяется встроенным образом в точке, где добавляется обработчик события. Для таких целей служит ключевое слово `delegate`.

14

Расширения в языке C#

В ЭТОЙ ГЛАВЕ...

- Применение инициализаторов
- Тип `var` и выведение типов
- Использование анонимных типов
- Тип `dynamic` и способы его применения
- Использование именованных и необязательных параметров
- Применение методов расширения
- Лямбда-выражения и способы их применения

Язык C# не стоит на месте. Андерс Хейлсберг (Anders Hejlsberg), автор языка C#, и другие разработчики из Microsoft постоянно обновляют и совершенствуют его. На момент написания настоящей книги самые последние изменения были включены в состав версии C# 4, которая является частью линейки продуктов Visual Studio 2010. К этому моменту может возникнуть вопрос, какие еще изменения могли понадобиться, ведь в предыдущих версиях C# действительно мало чего не доставало в плане функциональности. Однако разработчики языка C# сочли необходимым упростить некоторые аспекты программирования на C#, а также усовершенствовать отношения между C# и другими технологиями.

Пожалуй, наилучшим способом понять, что имеется в виду, будет вспомнить о дополнении, которое появилось между версиями 1.0 и 2.0 — *обобщения*. Хотя обобщения чрезвычайно полезны, никакой новой функциональности они на самом деле не предоставляют. Они, несомненно, значительно упрощают разработку, и без них пришлось бы писать гораздо больше кода. Вряд ли кто-то захотел бы вернуться во времена, когда еще не было обобщенных классов коллекций. Тем не менее, существенной частью C# обобщения не являются. С другой стороны, их, несомненно, можно считать улучшением языка.

Последующие языковые улучшения во многом схожи. Они предлагают новые способы для достижения результатов, которые в прошлом было трудно получить без применения объемного кода и/или сложных приемов программирования. В настоящей главе речь пойдет о нескольких таких улучшениях. О некоторых из них, например, о вариантности, уже рассказывалось ранее в этой книге.

Инициализаторы

В предшествующих главах было показано, как создавать экземпляры объектов и инициализировать их различными способами. В каждом случае для этого требовалось либо добавлять в определения классов дополнительный код для обеспечения инициализации, либо создавать экземпляры объектов и инициализировать их с помощью отдельных операторов. Кроме того, объяснялось, как создавать классы коллекций различных типов, в том числе обобщенные классы коллекций. Несложно было заметить, что простого способа для объединения создания коллекции с добавлением в нее элементов не предлагалось.

Инициализаторы объектов позволяют упростить код за счет объединения создания экземпляров и инициализации объектов. Инициализаторы коллекций предлагают простой и элегантный синтаксис для создания и заполнения коллекций за один шаг. В этом разделе будет показано, как использовать оба вида инициализаторов.

Инициализаторы объектов

Рассмотрим следующее простое определение класса:

```
public class Curry
{
  public string MainIngredient {get; set;}
  public string Style {get; set;}
  public int Spiciness {get; set;}
}
```

Этот класс имеет три свойства, которые определяются с применением показанного в главе 10 синтаксиса автоматических свойств. Если нужно создать и инициализировать экземпляр объекта этого класса, необходимо использовать несколько операторов:

```
Curry tastyCurry = new Curry();
tastyCurry.MainIngredient = "panir tikka";
tastyCurry.Style = "jalfrezi";
tastyCurry.Spiciness = 8;
```

Если в определение класса не включен конструктор, в коде будет вызываться конструктор по умолчанию без параметров, который предоставляется компилятором C#. Чтобы упростить эту инициализацию, можно снабдить класс соответствующим конструктором не по умолчанию:

```
public class Curry
{
  public Curry(string mainIngredient, string style, int spiciness)
  {
    MainIngredient = mainIngredient;
    Style = style;
    Spiciness = spiciness;
  }
  ...
}
```

Тогда можно написать следующий код, в котором создание экземпляра объединяется с инициализацией:

```
Curry tastyCurry = new Curry("panir tikka", "jalfrezi", 8);
```

Это работает нормально, хотя и принуждает код, работающий с данным классом, использовать именно этот конструктор, в результате чего предыдущий код, в котором применялся конструктор без параметров, не может быть выполнен. Зачастую, особенно в случаях, когда классы должны быть сериализируемыми, необходимо предоставлять также и конструктор без параметров:

```
public class Curry
{
  public Curry()
  {
  }
  ...
}
```

Теперь создавать и инициализировать экземпляр класса `Curry` можно любым способом. Однако чтобы добиться такого поведения, в первоначальное определение класса пришлось добавить несколько строк кода, которые кроме предоставления базовых деталей, необходимых для обеспечения подобной гибкости, больше ничего особо не делают.

Инициализаторы объектов предлагают способ создания и инициализации объектов без добавления какого-либо дополнительного кода (вроде описанного выше кода конструкторов). При создании экземпляра объекта необходимо предоставлять значения для общедоступных свойств или полей с использованием пар “имя-значение”. Ниже показан синтаксис:

```
<ИмяКласса> <имяПеременной> = new <ИмяКласса>
{
  <свойствоИлиПоле1> = <значение1>,
  <свойствоИлиПоле2> = <значение2>,
  ...
  <свойствоИлиПолеN> = <значениеN>
};
```

Приведенный выше код для создания и инициализация объекта типа `Curry` можно переписать следующим образом:

```
Curry tastyCurry = new Curry
{
  MainIngredient = "panir tikka",
  Style = "jalfrezi",
  Spiciness = 8
};
```

Часто код вроде этого размещают в одной строке без серьезного снижения читабельности.

При использовании инициализатора объекта не нужно явно вызывать конструктор класса. Если опустить круглые скобки конструктора (как в предыдущем коде), будет автоматически вызываться конструктор по умолчанию без параметров. Это происходит до установки инициализатором каких-либо значений параметров, что позволяет присвоить в конструкторе по умолчанию стандартные значения параметрам. В качестве альтернативы можно вызвать специфический конструктор. Так как он вызывается первым, любые устанавливаемые внутри него значения для общедоступных свойств можно переопределить, указав желаемые значения в инициализаторе. Для того чтобы инициализаторы значений работали, должен быть доступ к используемому конструктору (или к конструктору по умолчанию в случае неявного вызова).

Для инициализации с помощью объектного инициализатора свойства сложного типа можно применять *вложенный инициализатор объектов*. Синтаксис выглядит аналогично:

```
Curry tastyCurry = new Curry
{
  MainIngredient = "panir tikka",
  Style = "jalfrezi",
  Spiciness = 8,
  Origin = new Restaurant
  {
    Name = "King's Balti",
    Location = "York Road",
    Rating = 5
  }
};
```

Здесь осуществляется инициализация свойства `Origin` типа `Restaurant` (который тут не показан). В коде инициализируются три свойства `Origin` — `Name`, `Location` и `Rating` — значениями, соответственно, типа `string`, `string` и `int`. Именно так выглядит вложенный инициализатор объектов.

Обратите внимание, что объектные инициализаторы не являются заменой для конструкторов не по умолчанию объектов. При использовании объектных инициализаторов для установки значений свойств и полей во время создания экземпляра объекта не всегда известно, в какое состояние нужно инициализировать. С помощью конструкторов можно точно указать, какие значения требуются для того, чтобы объект мог функционировать, и затем немедленно выполнить код в ответ на эти значения.

Инициализаторы коллекций

В главе 5 было показано, как инициализировать массивы значениями с применением следующего синтаксиса:

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

Это быстрый и простой способ объединения создания и инициализации массива. Инициализаторы коллекций просто расширяют этот синтаксис для коллекций:

```
List<int> myIntCollection = new List<int> {5, 9, 10, 2, 99};
```

Комбинируя инициализаторы коллекций и объектов, можно конфигурировать коллекции с помощью простого и элегантного кода. Например, вместо кода:

```
List<Curry> curries = new List<Curry>();
curries.Add(new Curry("Chicken", "Pathia", 6));
curries.Add(new Curry("Vegetable", "Korma", 3));
curries.Add(new Curry("Prawn", "Vindaloo", 9));
```

можно использовать следующий код:

```
List<Curry> moreCurries = new List<Curry>
{
  new Curry
  {
    MainIngredient = "Chicken",
    Style = "Pathia",
    Spiciness = 6
  },
  new Curry
  {
    MainIngredient = "Vegetable",
    Style = "Korma",
    Spiciness = 3
  },
  new Curry
  {
    MainIngredient = "Prawn",
    Style = "Vindaloo",
    Spiciness = 9
  }
};
```

Такой подход очень хорошо подходит для типов, которые в основном используются для представления данных, и потому делает инициализаторы коллекций замечательным дополнением к технологии LINQ, которая будет описываться далее в книге.

Следующее практическое занятие демонстрирует работу с инициализаторами объектов и коллекций.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование инициализаторов

1. Создайте новое консольное приложение по имени `Ch14Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter14`.
2. Щелкните правой кнопкой мыши на имени этого проекта в окне Solution Explorer и выберите в контекстном меню пункт Add⇨Existing Item (Добавить⇨Существующий элемент).
3. Выберите файлы `Animal.cs`, `Cow.cs`, `Chicken.cs`, `SuperCow.cs` и `Farm.cs` из каталога `C:\BegVCSharp\Chapter12\Ch12Ex04\Ch12Ex04` и щелкните на кнопке Add (Добавить).
4. Измените объявление пространства имен в каждом из добавленных файлов следующим образом:

   ```
   namespace Ch14Ex01
   ```

5. Удалите конструкторы из классов `Cow`, `Chicken` и `SuperCow`.
6. Модифицируйте код в файле `Program.cs`, как показано ниже:

```
static void Main(string[] args)
{
  Farm<Animal> farm = new Farm<Animal>
  {
    new Cow { Name="Norris" },
    new Chicken { Name="Rita" },
    new Chicken(),
    new SuperCow { Name="Chesney" }
  };
```

```
    farm.MakeNoises();
    Console.ReadKey();
}
```

Фрагмент кода ***Ch14Ex01\Program.cs***

7. Выполните сборку приложения. На рис. 14.1 показаны ошибки, которые возникнут.

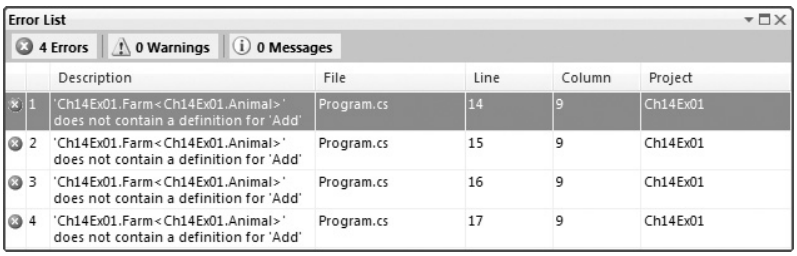


Рис. 14.1. *Ошибки при попытке выполнить сборку приложения*

8. Добавьте в файл `Farm.cs` следующий код:

```
public class Farm<T> : IEnumerable<T>
namespace Ch14Ex01
{
  public class Farm<T> : IEnumerable<T>
   where T : Animal
  {
    public void Add(T animal)
    {
      animals.Add(animal);
    }
   ...
```

Фрагмент кода ***Ch14Ex01\Farm.cs***

9. Запустите приложение. На рис. 14.2 показан результат, который должен получиться.

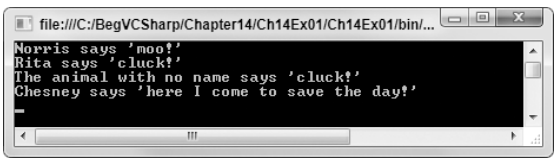


Рис. 14.2. *Приложение* `Ch14Ex01` *в действии*

Описание работы

В этом примере выполнялось создание и заполнение коллекции объектов за счет совместного использования инициализаторов объектов и инициализаторов коллекций. В роли коллекции выступила коллекция животных из предыдущих глав, но с двумя изменениями, которые необходимы для применения инициализаторов.

Во-первых, из всех унаследованных от `Animal` классов были удалены конструкторы. Причина в том, что в них устанавливается свойство `Name`, которое в примере будет устанавливаться с помощью инициализаторов объектов. В качестве альтернативы можно было бы добавить конструкторы по умолчанию. В любом случае при использовании конструктора по умолчанию свойство `Name` будет инициализироваться в соответствии с конструктором по умолчанию базового класса, код которого выглядит следующим образом:

```
public Animal()
{
  name = "The animal with no name";
}
```

При использовании инициализатора объекта с классом, унаследованным от `Animal`, важно помнить о том, что любые свойства, за установку которых отвечает инициализатор, устанавливаются после создания экземпляра объекта и, следовательно, после выполнения конструктора базового класса. Это означает, что значение, предоставляемое для свойства `Name` в инициализаторе объекта, будет переопределять значение, установленное для него по умолчанию. В рассматриваемом примере значение для свойства `Name` устанавливается для всех, кроме одного из добавляемых в коллекцию элементов.

Во-вторых, в класс `Farm` был добавлен метод `Add()`, чтобы отреагировать на ряд выданных компилятором ошибок следующего вида:

```
'Ch14Ex01.Farm <Ch14Ex01.Animal>' does not contain a definition for 'Add'
'Ch14Ex01.Farm <Ch14Ex01.Animal>' не содержит определения для 'Add'
```

Это ошибка отражает часть базовой функциональности инициализаторов коллекций. "За кулисами" компилятор вызывает метод `Add()` коллекции для каждого элемента, который предоставляется в инициализаторе коллекции. Класс `Farm` открывает доступ к коллекции объектов через свойство по имени `Animals`. Компилятор не может догадаться, что это и есть то самое свойство, которое нужно заполнить (методом `Animals.Add()`), и потому выдает сообщение об ошибке. Для устранения данной проблемы в класс `Farm` добавлен метод `Add()`, который инициализируется посредством инициализатора объектов.

В качестве альтернативы можно было бы изменить код в примере так, чтобы в нем предоставлялся вложенный инициализатор для свойства `Animals`:

```
static void Main(string[] args)
{
  Farm<Animal> farm = new Farm<Animal>
  {
    Animals =
    {
      new Cow { Name="Norris" },
      new Chicken { Name="Rita" },
      new Chicken(),
      new SuperCow { Name="Chesney" }
    }
  };
  farm.MakeNoises();
  Console.ReadKey();
}
```

Благодаря этому коду, предоставлять метод `Add()` для класса `Farm` не понадобилось бы. Такой вариант больше подходит для ситуаций, когда имеется класс, содержащий несколько коллекций. В данном случае очевидный кандидат на роль коллекции, в которую должны добавляться элементы с помощью метода `Add()` содержащего класса, отсутствует.

Выведение типов

Ранее в книге неоднократно упоминалось о том, что C# является *строго типизированным* языком, т.е. каждая переменная в нем имеет фиксированный тип и может использоваться только в том коде, в котором этот тип принимается во внимание. Во всех примерах, которые приводились до сих пор, переменные объявлялись с помощью такого кода:

```
<тип> <имяПеременной>;
```

или:

```
<тип> <имяПеременной> = <значение>;
```

В следующем коде сразу же понятно, к какому типу относится переменная `myInt`:

```
int myInt = 5;
Console.WriteLine(myInt);
```

IDE-среда тоже распознает тип переменной, в чем легко удостовериться, наведя курсор мыши на ее идентификатор (рис. 14.3).

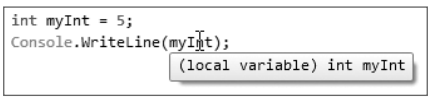


Рис. 14.3. *Определение типа переменной* `myInt` *в IDE-среде*

В C# 3 появилось новое ключевое слово `var`, которое можно использовать вместо метки *<тип>* в приведенном выше синтаксисе:

```
var <имяПеременной> = <значение>;
```

В этом коде переменная, указываемая на месте *<имяПеременной>*, *неявно* приводится к типу, заданному на месте *<значение>*. Обратите внимание, что никакого типа по имени `var` не существует. В следующей строке кода:

```
var myVar = 5;
```

под `myVar` подразумевается переменная типа `int`, а не переменная типа `var`. Как можно видеть на рис. 14.4, IDE-среда распознает и это.

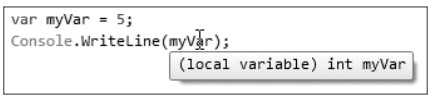


Рис. 14.4. *Определение типа переменной* `myVar` *в IDE-среде*

Этот момент чрезвычайно важен. Использование `var` вовсе не означает объявления переменной, не имеющей типа либо имеющей тип, который может изменяться. Если бы это было так, язык C# не являлся бы строго типизированным. Это означает лишь то, что ответственность за определение типа возлагается на компилятор.

Как будет показано в разделе "Динамический просмотр", появление динамических типов в .NET 4 расширяет определение C# как строго типизированного языка.

Если компилятору не удается определить тип переменной, объявленной с использованием `var`, код не скомпилируется. Следовательно, объявлять переменную с применением `var` и при этом не инициализировать ее не допускается, поскольку у компилятора не будет значения, которое он мог бы использовать для определения типа переменной. Это значит, что следующий код компилироваться не будет:

```
var myVar;
```

Ключевое слово `var` может также применяться и для выведения типа массива через его инициализатор:

```
var myArray = new[] {4, 5, 2};
```

В этом коде массиву myArray неявно назначается тип int[]. При неявной типизации массива элементы, используемые в инициализаторе, должны удовлетворять одному из перечисленных ниже правил:

- иметь один и тот же тип;
- иметь один и тот же ссылочный тип или null;
- представлять собой элементы, которые могут неявно приводиться к одному типу.

Если применяется последнее из этих правил, тип, к которому могут приводиться элементы, считается *наилучшим* типом для элементов массива. При появлении любой неоднозначности относительно выбора наилучшего типа, т.е. если есть два и более типов, к которым все элементы могут быть неявно приведены, код компилироваться не будет. В этом случае выдается сообщение об ошибке, информирующее о недоступности наилучшего типа:

```
var myArray = new[] {4, "not an int", 2};
```

Обратите внимание, что числовые значения никогда не интерпретируются как относящиеся к нулевым типам, и потому следующий код тоже не скомпилируется:

```
var myArray = new[] {4, null, 2};
```

Чтобы заставить его работать, можно воспользоваться стандартным инициализатором массива:

```
var myArray = new int?[] {4, null, 2};
```

И, наконец, следует помнить, что идентификатор var не запрещен к использованию в качестве имен классов. Это значит, например, что если в области действия (в том же или в ссылаемом пространстве имен) имеется класс по имени var, то неявно назначить тип с помощью ключевого слова var не удастся.

Сам по себе механизм выведения типов не особо полезен; скажем, в коде, который приводился ранее в данном разделе, он только все усложняет. Использование ключевого слова var затрудняет распознавание того, к какому типу относится та или иная переменная. Однако, как будет показано позже в главе, концепция выведения типов важна, поскольку лежит в основе других механизмов. Например, в анонимных типах, которые рассматриваются далее, он играет очень существенную роль.

Анонимные типы

При программировании, особенно приложений баз данных, часто оказывается, что много времени уходит на создание простых, скучных классов для представления данных. Семейства классов, не делающих больше ничего, кроме предоставления свойств, встречаются довольно часто. Показанный ранее в этой главе класс Curry является прекрасным тому примером:

```
public class Curry
{
  public string MainIngredient {get; set;}
  public string Style {get; set;}
  public int Spiciness {get; set;}
}
```

Этот класс фактически ничего не делает — он просто хранит структурированные данные. В базе данных или электронной таблице он мог бы просто представлять строку в таблице. Тогда класс коллекции, способный хранить экземпляры этого класса, мог бы служить представлением множества строк в этой таблице.

Такое применение классов вполне допустимо, однако написание кода для этих классов может превратиться в рутинную работу, а внесение любых изменений в лежащую в основе схему данных потребует добавления, удаления или изменения определяющего их кода.

Анонимные типы — это способ упрощения этой модели программирования. Идея состоит в том, чтобы не определять такие простые типы для хранения данных, а позволить компилятору C# автоматически создавать типы на основе данных, которые должны в них храниться.

Экземпляр показанного выше типа `Curry` создается следующим образом:

```
Curry curry = new Curry
{
  MainIngredient = "Lamb",
  Style = "Dhansak",
  Spiciness = 5
};
```

В качестве альтернативы можно воспользоваться анонимным типом:

```
var curry = new
{
  MainIngredient = "Lamb",
  Style = "Dhansak",
  Spiciness = 5
}
```

Между этими фрагментами кода есть два отличия. Во-первых, используется ключевое слово `var`. Объясняется это тем, что анонимные типы не имеют идентификатора, который можно было бы использовать. Внутренне они все-таки получают идентификатор, как будет показано ниже, однако работать с ним в коде нельзя. Во-вторых, после ключевого слова `new` никакого имени для типа не указано. Именно так компилятор узнает, что необходимо использовать анонимный тип.

IDE-среда распознает определение анонимного типа и соответствующим образом обновляет информацию IntelliSense. Для приведенного выше объявления информация об анонимном типе будет выглядеть так, как показано на рис. 14.5.

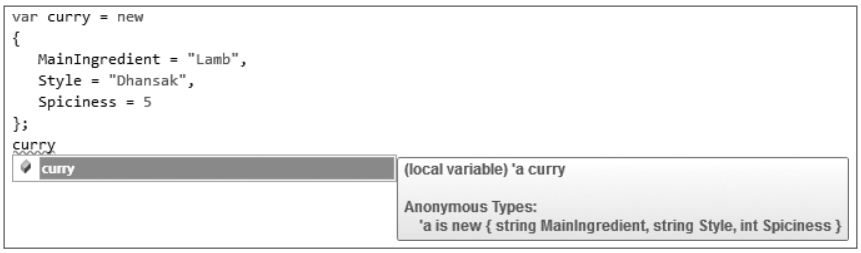


Рис. 14.5. *Отображение информации об анонимном типе в IntelliSense*

Здесь видно, что на внутреннем уровне переменная `curry` относится к типу `'a`. Очевидно, что использовать такой тип в коде нельзя, поскольку его имя не является допустимым для идентификатора. С помощью символа `'` средство IntelliSense обозначает анонимные типы. Кроме того, в IntelliSense можно просматривать члены анонимного типа, как показано на рис. 14.6.

Обратите внимание, что свойства анонимных типов являются *доступными только для чтения*. Это значит, что если нужно иметь возможность изменять значения свойств в объектах для хранения данных, то применять анонимные типы нельзя.

Реализации остальных членов анонимных типов посвящено следующее практическое занятие.

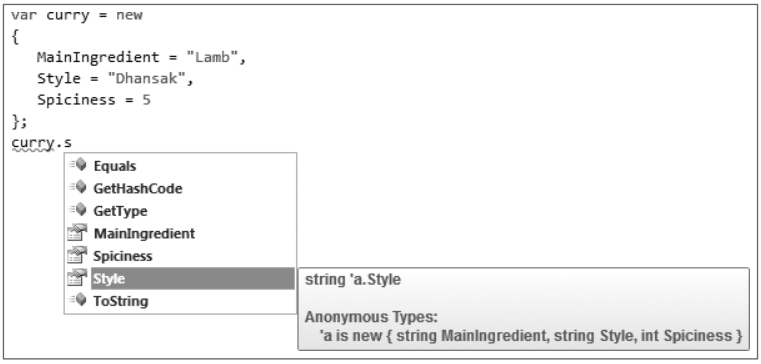


Рис. 14.6. *Отображение информации о членах анонимного типа в IntelliSense*

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование анонимных типов

1. Создайте новое консольное приложение по имени `Ch14Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter14`.
2. Измените код в его файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  var curries = new[]
  {
    new
    {
      MainIngredient = "Lamb",
      Style = "Dhansak",
      Spiciness = 5
    },
    new
    {
      MainIngredient = "Lamb",
      Style = "Dhansak",
      Spiciness = 5
    },
    new
    {
      MainIngredient = "Chicken",
      Style = "Dhansak",
      Spiciness = 5
     }
  };
  Console.WriteLine(curries[0].ToString());
  Console.WriteLine(curries[0].GetHashCode());
  Console.WriteLine(curries[1].GetHashCode());
  Console.WriteLine(curries[2].GetHashCode());
  Console.WriteLine(curries[0].Equals(curries[1]));
  Console.WriteLine(curries[0].Equals(curries[2]));
  Console.WriteLine(curries[0] == curries[1]);
  Console.WriteLine(curries[0] == curries[2]);
  Console.ReadKey();
}
```

Фрагмент кода ***Ch14Ex02\Program.cs***

3. Запустите приложение. На рис. 14.7 показан результат, который должен получиться.

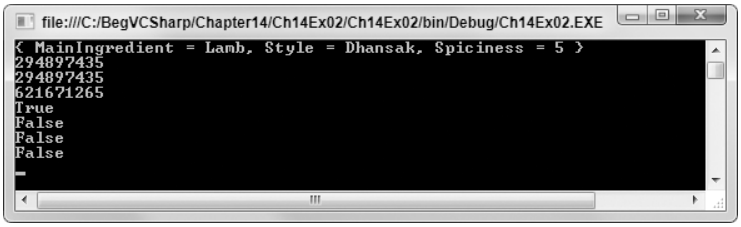


Рис. 14.7. *Приложение* `Ch14Ex02` *в действии*

Описание работы

В этом примере создается массив объектов с анонимными типами, который затем используется для проверки членов анонимных типов. Код для создания этого массива выглядит следующим образом:

```
var curries = new[]
{
  new
  {
    MainIngredient = "Lamb",
    Style = "Dhansak",
    Spiciness = 5
  },
  ...
};
```

Здесь используется массив, который неявно приводится к анонимному типу за счет применения комбинированного синтаксиса, одна часть которого демонстрировалась в этом разделе, а другая — в разделе "Выведение типов" ранее в главе. В результате переменная `curries` содержит три экземпляра анонимного типа.

После создания такого массива в коде первым делом производится отображение в окне консоли результата вызова метода `ToString()` на анонимном типе:

```
Console.WriteLine(curries[0].ToString());
```

Это дает следующий вывод:

```
{ MainIngredient = Lamb, Style = Dhansak, Spiciness = 5 }
```

Реализация метода `ToString()` в анонимном типе предусматривает вывод значений каждого из определенных для этого типа свойств.

Далее в коде производится вызов метода `GetHashCode()` на каждом из трех объектов массива:

```
Console.WriteLine(curries[0].GetHashCode());
Console.WriteLine(curries[1].GetHashCode());
Console.WriteLine(curries[2].GetHashCode());
```

В случае реализации метод `GetHashCode()` должен возвращать для объекта уникальное целое число на основе его состояния. Первые два объекта в массиве содержат одни и те же значения в свойствах и, следовательно, обладают одинаковым состоянием. Поэтому результаты вызова `GetHashCode()` для первых двух совпадают. Весь вывод показан ниже:

```
294897435
294897435
621671265
```

Затем производится вызов метода `Equals()` для сравнения первого объекта со вторым и с третьим объектом:

```
Console.WriteLine(curries[0].Equals(curries[1]));
Console.WriteLine(curries[0].Equals(curries[2]));
```

Это дает следующий результат:

```
True
False
```

Реализация метода `Equals()` в анонимных типах сравнивает состояния объектов. Значение `true` возвращается, когда значения всех свойств одного объекта в точности совпадают со значениями свойств другого объекта.

Однако при использовании операции == такого не происходит. Операция ==, как было показано в предыдущих главах, сравнивает объектные ссылки. В последнем разделе кода производятся те же сравнения, что и ранее, но с применением операции == вместо метода `Equals()`:

```
Console.WriteLine(curries[0] == curries[1]);
Console.WriteLine(curries[0] == curries[2]);
```

Каждый из элементов в массиве `curries` ссылается на разный экземпляр анонимного типа, поэтому в результате в обоих случаях возвращается значение `false`. Вывод, как и следовало ожидать, получается таким:

```
False
False
```

Интересно, что при создании экземпляров анонимных типов компилятор заметил, что параметры выглядят одинаково, и создал три экземпляра *одного и того же* анонимного типа, а не трех отдельных анонимных типов. Тем не менее, это не означает, что при создании экземпляра объекта из анонимного типа компилятор выполняет поиск совпадающего типа. Даже в случае определения где-то в коде класса с соответствующими свойствами, при использовании синтаксиса анонимных типов всегда будет создаваться (или, как было в рассмотренном примере, повторно использоваться) анонимный тип.

Динамический просмотр

Ключевое слово `var`, как упоминалось ранее, само типом не является, следовательно, "строго типизированной" методологии языка C# оно не нарушает. В версии C# 4, однако, вещи стали менее фиксированными. В C# 4 появилась концепция *динамических переменных*, каковые, как не трудно догадаться по их названию, представляют собой переменные, не имеющие фиксированного типа.

Главным стимулом для добавления таких переменных послужил тот факт, что во многих ситуациях необходимо использовать C# для манипуляций с объектами, которые были созданы на каком-то другом языке. К ним относится взаимодействие с более старыми технологиями наподобие COM (Component Object Model — модель компонентных объектов), а также работа с динамическими языками, такими как JavaScript, Python и Ruby. Для доступа к методам и свойствам объектов, созданных с помощью этих языков, ранее в C# использовался довольно громоздкий синтаксис. Например, предположим, что имеется код, извлекающий JavaScript-объект с методом `Add()`, который складывает два числа. Без технологии динамического просмотра код C# мог бы выглядеть примерно так:

```
ScriptObject jsObj = SomeMethodThatGetsTheObject();
int sum = Convert.ToInt32(jsObj.Invoke("Add", 2, 3));
```

Тип `ScriptObject` (подробно здесь не рассматривается) предоставляет доступ к JavaScript-объекту, но даже он не способен обеспечить возможность делать следующее:

```
int sum = jsObj.Add(2, 3);
```

Технология динамического просмотра (dynamic lookup) меняет все и позволяет писать код, подобный показанному выше. Однако, как будет показано в последующих разделах, за такую мощь приходится платить свою цену.

Еще одной ситуацией, в которой технология динамического просмотра может оказаться полезной, является необходимость иметь дело с объектом C#, тип которого не известен. Хоть ситуация выглядит странной, возникает она гораздо чаще, чем может показаться. Также динамический просмотр важен при написании обобщенного кода, способного иметь дело с любыми получаемыми входными данными. "Старый" способ предполагал применение *рефлексии*, при которой для доступа к типам и их членам использовалась информация о типах. Синтаксис рефлексии на самом деле во многом похож на тот, что применялся для получения доступа к JavaScript-объекту.

"За кулисами" динамический просмотр поддерживается компонентом DLR (Dynamic Language Runtime — исполняющая среда динамического языка), который входит в состав .NET 4 точно так же, как CLR (Common Language Runtime — общеязыковая исполняющая среда). Полное описание DLR выходит за рамки настоящей книги.

Тип `dynamic`

В C# появилось новое ключевое слова `dynamic`, которое можно использовать для определения переменных, например:

```
dynamic myDynamicVar;
```

В отличие от ключевого слова `var`, тип `dynamic` действительно существует, поэтому инициализировать значение переменной `myDynamicVar` при ее объявлении не обязательно.

Необычно то, что тип `dynamic` существует только во время компиляции; на этапе выполнения вместо него используется тип `System.Object`. Данная особенность реализации не является особо существенной, но запомнить ее стоит, поскольку это позволит лучше понять некоторые из приведенных ниже обсуждений. После создания динамической переменной можно получать доступ к ее членам (код для получения значения переменной здесь не показан):

```
myDynamicVar.DoSomething("With this!");
```

Какое бы значение в действительности не содержалось в переменной `myDynamicVar`, этот код все равно будет компилироваться. Однако если запрашиваемого члена не существует, во время выполнения будет выдаваться исключение типа `RuntimeBinderException`.

По сути, этот код просто предоставляет "рецепт", который должен применяться во время выполнения. Во время выполнения просматривается значение переменной `myDynamicVar`, затем обнаруживается и при необходимости вызывается метод по имени `DoSomething()`.

В следующем практическом занятии будет рассмотрен конкретный пример.

Следующий пример предназначен исключительно для демонстративных целей! Обычно динамические типы должны использоваться, только если никакие другие варианты не доступны, например, при работе с объектами, не относящимися к .NET.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование динамических типов

1. Создайте новое консольное приложение по имени `Ch14Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter14`.

2. Измените код в его файле `Program.cs` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CSharp.RuntimeBinder;

namespace Ch14Ex03
{
  class MyClass1
  {
    public int Add(int var1, int var2)
    {
      return var1 + var2;
    }
  }
  class MyClass2
  {
  }
  class Program
  {
    static int callCount = 0;
    static dynamic GetValue()
    {
      if (callCount++ == 0)
      {
        return new MyClass1();
      }
      return new MyClass2();
    }
    static void Main(string[] args)
    {
      try
      {
        dynamic firstResult = GetValue();
        dynamic secondResult = GetValue();
        Console.WriteLine("firstResult is: {0}", firstResult.ToString());
                          // Вывод значения firstResult
        Console.WriteLine("secondResult is: {0}", secondResult.ToString());
                          // Вывод значения firstResult
        Console.WriteLine("firstResult call: {0}", firstResult.Add(2, 3));
                          // Вызов метода Add() на firstResult
        Console.WriteLine("secondResult call: {0}", secondResult.Add(2, 3));
                          // Вызов метода Add() на secondResult
      }
      catch (RuntimeBinderException ex)
      {
        Console.WriteLine(ex.Message);
      }
      Console.ReadKey();
    }
  }
}
```

Фрагмент кода ***Ch14Ex03\Program.cs***

3. Запустите приложение. На рис. 14.8 показан результат, который должен получиться.

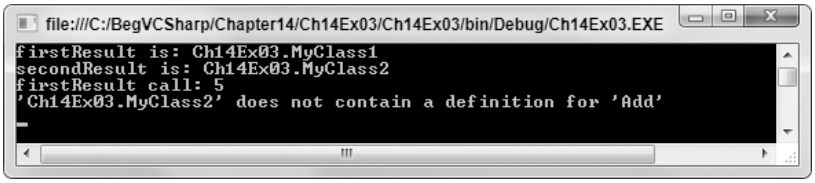


Рис. 14.8. *Приложение* `Ch14Ex03` *в действии*

Описание работы

В этом примере сначала используется метод, который возвращает один из двух типов объектов, для получения динамического значения, а затем предпринимается попытка использовать полученные объекты. Код компилируется без проблем, но при попытке доступа к несуществующему методу выдается (и обрабатывается) исключение.

Вначале добавляется оператор `using` для пространства имен, в котором содержится `RuntimeBindingException`:

```
using Microsoft.CSharp.RuntimeBinder;
```

Затем определяются два класса `MyClass1` и `MyClass2`, причем класс `MyClass1` имеет метод `Add()`, а класс `MyClass2` не содержит членов:

```
class MyClass1
{
  public int Add(int var1, int var2)
  {
    return var1 + var2;
  }
}
class MyClass2
{
}
```

Кроме того, в класс `Program` добавлено поле (`callCount`) и метод (`GetValue()`), которые позволяют получить экземпляр одного из этих классов:

```
static int callCount = 0;
static dynamic GetValue()
{
  if (callCount++ == 0)
  {
    return new MyClass1();
  }
  return new MyClass2();
}
```

Здесь используется простой счетчик вызовов, чтобы этот метод возвращал экземпляр `MyClass1` при его первом вызове и экземпляры `MyClass2` — в последующих. Обратите внимание, что ключевое слово `dynamic` может использоваться в качестве возвращаемого типа метода.

Далее в коде `Main()` дважды вызывается метод `GetValue()` и производится попытка вызвать `ToString()` и `Add()` в отношении обоих значений. Этот код помещен в блок `try...catch` для перехвата любых исключений типа `RuntimeBinderException`, которые могут возникнуть.

```
static void Main(string[] args)
{
  try
  {
    dynamic firstResult = GetValue();
    dynamic secondResult = GetValue();
    Console.WriteLine("firstResult is: {0}",
      firstResult.ToString());
    Console.WriteLine("secondResult is: {0}",
      secondResult.ToString());
    Console.WriteLine("firstResult call: {0}",
      firstResult.Add(2, 3));
    Console.WriteLine("secondResult call: {0}",
      secondResult.Add(2, 3));
  }
  catch (RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
  Console.ReadKey();
}
```

Несомненно, при вызове `secondResult.Add()` будет сгенерировано исключение, потому что такого метода в `MyClass` не существует. В сообщении об исключении именно на это будет указано.

Ключевое слово `dynamic` может также использоваться в других местах, где требуется имя типа, например, в параметрах метода. Метод `Add()` можно было бы переписать следующим образом:

```
public int Add(dynamic var1, dynamic var2)
{
   return var1 + var2;
}
```

На результат это никак не повлияло бы. В таком случае во время выполнения сначала производится изучение передаваемых `var1` и `var2` значений, чтобы выяснить, существует ли совместимое определение операции `+`. Если передаются два значения `int`, такая операция существует. Для несовместимых значений генерируется исключение `RuntimeBinderException`. Например, попытка выполнить следующую строку кода:

```
Console.WriteLine("firstResult call: {0}", firstResult.Add("2", 3));
```

приводит к выдаче исключения со следующим сообщением:

```
Cannot implicitly convert type 'string' to 'int'
Не удается неявно преобразовать тип 'string' в 'int'
```

Вывод из всего этого таков: динамические типы являются очень мощными, однако с ними связаны и некоторые ограничения. Описанных исключений можно полностью избежать, применяя строгую типизацию вместо динамической. В большей части разрабатываемого кода C# использования ключевого слова `dynamic` можно избегать. Когда в нем действительно возникает необходимость, оно превращается в достаточно мощный инструмент.

Интерфейс `IDynamicMetaObjectProvider`

Прежде чем двигаться дальше, не помешает узнать о том, как используются динамические типы, точнее — что происходит, когда во время выполнения применяется "рецепт" для доступа к членам. В действительности существуют три варианта поведения.

- Если динамическое значение представляет собой объект COM, то для получения доступа к членам применяются технологии COM (через интерфейс `IUnknown`, хотя знать об этом здесь не обязательно).
- Если динамическое значение поддерживает интерфейс `IDynamicMetaObjectProvider`, то для получения доступа к членам типа применяется этот интерфейс.
- Если не подходит ни первый, ни второй вариант, то применятся рефлексия.

Наиболее интересным здесь является второй случай, который предусматривает применение интерфейса `IDynamicMetaObjectProvider`. Не вдаваясь в детали, следует отметить, что данный интерфейс можно реализовать для управления тем, что в точности должно происходить при получении доступа к членам типа во время выполнения. Эта сложная тема в книге не рассматривается.

Расширенные параметры методов

Язык C# 4 предлагает более широкие возможности в плане определения и использования параметров методов. Это главным образом вызвано специфической проблемой, которая возникает при использовании интерфейсов, которые определены внешне, вроде модели программирования Microsoft Office. Здесь некоторые методы имеют огромное количество параметров, многие из которых не являются обязательными для каждого вызова. Раньше нужно было заботиться либо о способе указания недостающих параметров, либо добавлять в вызов массу значений `null`:

```
RemoteCall(var1, var2, null, null, null, null, null);
```

В этом коде совершенно не очевидно, на что ссылаются значения `null`, или почему они были пропущены.

В идеальном случае могло бы существовать множество перегруженных версий данного метода `RemoteCall()`, в том числе и такая, которая бы принимала лишь два параметра, как показано ниже:

```
RemoteCall(var1, var2);
```

Однако это потребовало бы наличия множества других методов с альтернативными комбинациями параметров, что привело бы к дополнительным проблемам (вроде необходимости написания большего объема кода, увеличения его сложности и т.д.).

В языках вроде Visual Basic эта ситуация была разрешена иным образом, за счет возможности использовать именованные и необязательные параметры. В C# 4 также появилась такая возможность, что демонстрирует тенденцию к сближению всех языков .NET.

В следующих разделах показано, как использовать в C# параметры нового вида.

Необязательные параметры

Часто в нескольких вызовах метода определенному параметру передается одно и то же значение. Это может быть, например, булевское значение, отвечающее за управление какой-то несущественной частью функциональности метода. Рассмотрим следующий метод:

```
public List<string> GetWords(
  string sentence,
  bool capitalizeWords)
{
  ...
}
```

Какое бы значение не передавалось параметру `capitalizeWords`, этот метод всегда будет возвращать список значений `string`, представляющих слова из переданного в качестве входных данных предложения. Иногда нужно, чтобы первые буквы слов в списке были

преобразованы в заглавные, но в большинстве случаев это не требуется, и вызов метода выглядит следующим образом:

```
List<string> words = GetWords(sentence, false);
```

Чтобы сделать это поведением по умолчанию, можно объявить второй метод следующим образом:

```
public List<string> GetWords(string sentence)
{
  return GetWords(sentence, false);
}
```

Ничего неправильного в этом нет, но, как нетрудно представить, в ситуации с большим количеством параметров это может превратиться в настоящий кошмар.

В качестве альтернативы можно сделать параметр `capitalizeWords` *необязательным*. Для этого в определении метода для параметра `capitalizeWords` необходимо предоставить значение по умолчанию, которое будет использоваться, если в вызове для `capitalizeWords` значение не указано:

```
public List<string> GetWords(
  string sentence,
  bool capitalizeWords = false)
{
  ...
}
```

При таком определении методу можно передавать либо один, либо два параметра, причем второй параметр обязателен, только если необходимо присвоить `capitalizeWords` значение `true`.

Значения необязательных параметров

Как было описано в предыдущем разделе, необязательный параметр задается в определении метода с помощью такого синтаксиса:

<типПараметра> <имяПараметра> = <значениеПоУмолчанию>

В отношении значения, указываемого на месте *<значениеПоУмолчанию>*, существуют некоторые ограничения. Значения по умолчанию должны быть литеральными значениями, константными значениями, новыми экземплярами объектов либо значениями используемого по умолчанию типа значения. Это значит, что следующий код компилироваться не будет:

```
public bool CapitalizationDefault;

public List<string> GetWords(
  string sentence,
  bool capitalizeWords = CapitalizationDefault)
{
...
}
```

Чтобы заставить этот код работать, значение `CapitalizationDefault` понадобится определить как константное:

```
public const bool CapitalizationDefault = false;
```

Имеет ли смысл делать подобное, зависит от ситуации; в большинстве случаев предпочтение отдается использованию литерального значения.

Порядок необязательных параметров

Необязательные значения должны находиться в конце списка параметров для метода. После параметров со значениями по умолчанию не могут следовать параметры без таких значений.

Приведенный ниже код является недопустимым:

```
public List<string> GetWords(
  bool capitalizeWords = false,
  string sentence)
  {
    ...
  }
```

Здесь sentence представляет собой обязательный параметр и потому должен находиться перед необязательным параметром capitalizedWords.

Именованные параметры

При использовании необязательных параметров могут возникать ситуации, когда определенный метод имеет несколько необязательных параметров. При этом необходимо предоставлять значение, например, только третьему параметру. Приведенный ранее синтаксис не позволяет это сделать, не передавая значения также для первого и второго необязательного параметра.

По этой причине в C# 4 появились *именованные параметры*. Они не требуют что-либо менять в определении метода, а представляют собой прием, применяемый при вызове метода. Необходимый синтаксис показан ниже:

```
MyMethod(
  <имяПараметра>: <значениеПараметра>,
  ...
  <имяПараметра>: <значениеПараметра>);
```

Имена параметров — это имена переменных, которые используются в определении метода.

Подобным образом можно указывать любое количество параметров, причем в любом порядке. Кроме того, именованные параметры могут быть необязательными.

При желании именованные параметры можно использовать только для некоторых параметров в вызове метода. Это удобно, когда в сигнатуре метода есть несколько необязательных параметров, а также ряд обязательных. В таком случае можно сначала указать обязательные параметры, а после них — все нужные именованные необязательные параметры, например:

```
MyMethod(
  requiredParameter1Value,
  optionalParameter5: optionalParameter5Value);
```

Смешивая именованные и позиционные параметров, обратите внимание, что все позиционные параметры должны быть предоставлены первыми, перед именованными параметрами. Если именованные параметры применяются везде, при желании в вызове можно использовать другой порядок параметров, например:

```
MyMethod(
  optionalParameter5: optionalParameter5Value,
  requiredParameter1: requiredParameter1Value);
```

В этом случае для всех обязательных параметров должны быть предоставлены значения.

Следующее практическое занятие посвящено использованию именованных и необязательных параметров.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование необязательных и именованных параметров

1. Создайте новое консольное приложение по имени `Ch14Ex04` и сохраните его в каталоге `C:\BegVCSharp\Chapter14`.
2. Добавьте в него класс по имени `WordProcessor` и модифицируйте код внутри него следующим образом:

```
public static class WordProcessor
{
  public static List<string> GetWords(
    string sentence,
    bool capitalizeWords = false,
    bool reverseOrder = false,
    bool reverseWords = false)
  {
    List<string> words = new List<string>(sentence.Split(' '));
    if (capitalizeWords)
      words = CapitalizeWords(words);
    if (reverseOrder)
      words = ReverseOrder(words);
    if (reverseWords)
      words = ReverseWords(words);
    return words;
  }
  private static List<string> CapitalizeWords(
    List<string> words)
  {
    List<string> capitalizedWords = new List<string>();
    foreach (string word in words)
    {
       if (word.Length == 0)
         continue;
       if (word.Length == 1)
        capitalizedWords.Add(
         word[0].ToString().ToUpper());
      else
       capitalizedWords.Add(
        word[0].ToString().ToUpper()
         + word.Substring(1));
    }
    return capitalizedWords;
  }
  private static List<string> ReverseOrder(List<string> words)
  {
    List<string> reversedWords = new List<string>();
    for (int wordIndex = words.Count - 1;
      wordIndex >= 0; wordIndex--)
     reversedWords.Add(words[wordIndex]);
    return reversedWords;
  }
  private static List<string> ReverseWords(List<string> words)
  {
    List<string> reversedWords = new List<string>();
    foreach (string word in words)
     reversedWords.Add(ReverseWord(word));
    return reversedWords;
  }
```

```
  private static string ReverseWord(string word)
  {
    StringBuilder sb = new StringBuilder();
    for (int characterIndex = word.Length - 1;
        characterIndex >= 0; characterIndex--)
    sb.Append(word[characterIndex]);
    return sb.ToString();
  }
}
```

Фрагмент кода ***Ch14Ex04\WordProcessor.cs***

3. Измените код в файле `Program.cs`, как показано ниже:

```
static void Main(string[] args)
{
  string sentence = "'twas brillig, and the slithy toves did gyre "
    + "and gimble in the wabe:";
  List<string> words;

  words = WordProcessor.GetWords(sentence);
  Console.WriteLine("Original sentence:");
                    //Исходное предложение
  foreach (string word in words)
  {
    Console.Write(word);
    Console.Write(' ');
  }
  Console.WriteLine('\n');

  words = WordProcessor.GetWords(
    sentence,
    reverseWords: true,
    capitalizeWords: true);

  Console.WriteLine("Capitalized sentence with reversed words:");
                  // То же предложение, но со словами, начинающимися
                  // с заглавных букв и в обратном порядке
  foreach (string word in words)
  {
     Console.Write(word);
     Console.Write(' ');
  }
  Console.ReadKey();
}
```

Фрагмент кода ***Ch14Ex04\Program.cs***

4. Запустите приложение. На рис. 14.9 показан результат, который должен получиться.

Рис. 14.9. *Приложение* `Ch14Ex04` *в действии*

Описание работы

В этом примере был создан вспомогательный класс, способный выполнять простые манипуляции со строками, который использовался для изменения строки. Единственный общедоступный метод этого класса принимает один обязательный и три необязательных параметра:

```
public static List<string> GetWords(
  string sentence,
  bool capitalizeWords = false,
  bool reverseOrder = false,
  bool reverseWords = false)
{
  ...
}
```

Фрагмент кода ***Ch14Ex04\WordProcessor.cs***

Этот метод возвращает коллекцию значений `string`, каждое из которых представляет собой слово из исходного предложения. В зависимости от того, какие из трех необязательных параметров указаны (если указаны), он может выполнять дополнительные трансформации как над коллекцией строк в целом, так и над отдельными значениями-словами.

Более подробно функциональность класса `WordProcessor` здесь рассматриваться не будет; желающие могут самостоятельно изучить его код. Можно также подумать о путях его совершенствования, например, должно ли слово `'twas` действительно преобразовываться в `'Twas`, и как внести такое изменение.

При вызове этого метода используется только два из доступных необязательных параметров, третий параметр (`reverseOrder`) получает свое значение по умолчанию, равное `false`:

```
words = WordProcessor.GetWords(
  sentence,
  reverseWords: true,
  capitalizeWords: true);
```

Обратите внимание, что два параметра, которые используются, указаны не в том порядке, в котором были определены.

И, наконец, следует отметить удобство средства IntelliSense при работе с методами, которые имеют необязательные параметры. При вводе кода последнего практического занятия для метода `GetWords()` должна появиться подсказка, показанная на рис. 14.10 (которая также отображается после наведения курсора мыши на имя метода).

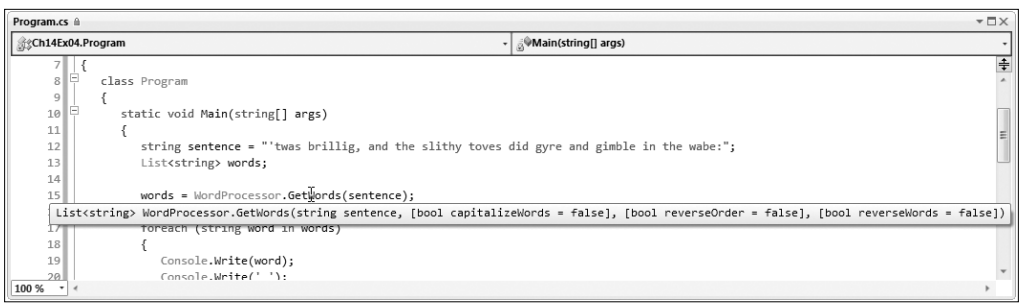


Рис. 14.10*. Работа средства IntelliSense для метода с необязательными параметрами*

Это очень полезная подсказка, поскольку в ней отображаются не только имена доступных параметров, но и выбранные для необязательных параметров значения по умолчанию, что позволяет сразу решить, нужно ли переопределять какое-то из них.

Рекомендации относительно именованных и необязательных параметров

Появление именованных и необязательных параметров было встречено неоднозначно. Одних разработчиков, в частности тех, которые имеют дело с Microsoft Office, их появление очень обрадовало. Однако многим другим они показались ненужным изменением языка C#, поскольку, по их мнению, хорошо определенный пользовательский интерфейс не должен нуждаться в подобных средствах доступа, по крайней мере, не до такой степени, чтобы вносить изменения в сам язык.

Строго говоря, именованные и необязательные параметры обладают рядом полезных достоинств, но чрезмерное их использование может оказать пагубное влияние на код. В некоторых ситуациях, как в упомянутом сценарии с Microsoft Office, они, несомненно, полезны. Кроме того, иногда они упрощают код, как было показано в последнем практическом занятии, где определялось множество вариантов для управления функционированием метода. Но в большинстве случаев применение именованных и необязательных параметров без наличия веских оснований нельзя назвать хорошей идеей. Неплохая проверка состоит в том, чтобы по вызову метода попробовать определить, каким может быть результат, не зная заранее, что делает метод. Если параметры и способ их использования очевидны (как и должно быть в хорошо написанном коде), применять именованные и/или необязательные параметры не имеет смысла.

Методы расширения

Методы расширения (extension methods) — это способ расширения функциональности типов без модификации самих типов. Их можно использовать для расширения даже тех типов, которые изменять невозможно, включая типы, определенные в .NET Framework. Например, применяя метод расширения, можно снабдить дополнительной функциональностью даже такой фундаментальный тип, как `System.String`.

В данном контексте под *расширением функциональности* типа понимается предоставление такого метода, который мог бы вызываться через экземпляр этого типа. Создаваемый для этого метод, который и называется *методом расширения*, может принимать любое количество параметров и возвращать значение любого типа (в том числе `void`). Чтобы создать и использовать метод расширения, необходимо выполнить следующие действия.

1. Создать необобщенный статический класс.
2. Добавить в этот класс метод расширения в виде статического метода, воспользовавшись специально предназначенным для этого синтаксисом (рассматривается чуть ниже).
3. В коде, где планируется использовать метод расширения, с помощью оператора `using` импортировать пространство имен, содержащее класс с этим методом.
4. Вызвать метод расширения через экземпляр расширенного типа таким же образом, как любой другой метод этого типа.

Компилятор C# принимает участие в третьем и четвертом действиях, благодаря чему IDE-среда мгновенно распознает, что был создан метод расширения, и даже отображает его в списке IntelliSense (рис. 14.11).

***Рис. 14.11**. Отображение средством IntelliSense информации о методе расширения*

На рис. 14.11 видно, что метод расширения, который здесь называется `MyMarvelous ExtensionMethod()`, доступен через объект `string` (в данном случае это просто литеральная строка). Этот метод, который помечен несколько отличающимся значком с указывающей вниз стрелкой синего цвета, не принимает дополнительных параметров и возвращает `string`.

Метод расширения определяется подобно любому другому методу, но при этом нужно соблюдать требования синтаксиса, предусмотренного специально для методов расширения.

- Метод должен обязательно быть статическим.
- Метод должен обязательно включать в себя параметр, представляющий экземпляр типа, в отношении которого он будет вызываться. (Здесь этот параметр будет называться *параметром экземпляра*.)
- Параметр экземпляра должен обязательно определяться в методе первым.
- Параметр экземпляра не должен сопровождаться больше никаким другим модификатором, кроме ключевого слова `this`.

Ниже показан синтаксис для определения метода расширения:

```
public static class ExtensionClass
{
  public static <ВозвращаемыйТип> <ИмяМетодаРасширения>(
    this <РасширяемыйТип> instance)
 {
    ...
  }
}
```

После импорта пространства имен, внутри которого содержится класс, включающий в себя данный метод (это называется сделать метод расширения доступным), можно переходить к написанию кода:

```
<РасширяемыйТип> myVar;
// myVar инициализируется в коде, который здесь не показан.
myVar.<ИмяМетодаРасширения>();
```

При желании в метод расширения еще можно добавлять любые параметры и использовать возвращаемый тип.

Приведенный выше вызов идентичен следующему, но имеет более простой синтаксис:

```
<ПодлежащийРасширениюТип> myVar;
// myVar инициализируется в коде, который здесь не показан.
ExtensionClass.<ИмяМетодаРасширения>(myVar);
```

Другое преимущество состоит в том, что после выполнения импорта намного упрощается поиск необходимой функциональности за счет просмотра информации о методах расширения в IntelliSense. Методы расширения могут быть разбросаны по многим классам расширений или даже библиотекам, но все они будут появляться в списке членов расширенного типа.

После определения метода расширения, который может использоваться с конкретным типом, его можно также использовать с любым унаследованным от него типом. Если вспомнить приводившийся ранее в этой главе пример с животными, то определение метода расширения для класса `Animal` позволило бы вызывать его также, например, и в отношении объекта `Cow`.

Можно также определять методы расширения, которые функционируют на конкретном интерфейсе, и затем использовать их для любого типа, который реализует этот интерфейс.

Методы расширения — замечательный способ для предоставления библиотек вспомогательного кода, пригодного для многократного использования во множестве различных приложений. Они также широко применяются в технологии LINQ, которая более подробно рассматривается позже в книге. Следующее практическое занятие посвящено работе с методами расширения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Определение и использование методов расширения

1. Создайте новое консольное приложение по имени `Ch14Ex05` и сохраните его в каталоге `C:\BegVCSharp\Chapter14`.
2. Добавьте в решение новый проект библиотеки классов по имени `ExtensioLib`.
3. Удалите из `ExtensionLib` существующий файл класса `Class1.cs` и добавьте в него файл класса `WordProcessor.cs` из предыдущего примера `Ch14Ex04`.
4. Измените код в файле `WordProcessor.cs` следующим образом:

```
namespace ExtensionLib
{
  public static class WordProcessor
  {
    public static List<string> GetWords(
      this string sentence,
      bool capitalizeWords = false,
      bool reverseOrder = false,
      bool reverseWords = false)
    {
      ...
    }
    ...
    public static string ToStringReversed(this object inputObject)
    {
      return ReverseWord(inputObject.ToString());
    }
    public static string AsSentence(this List<string> words)
    {
      StringBuilder sb = new StringBuilder();
      for (int wordIndex = 0; wordIndex < words.Count; wordIndex++)
    {
      sb.Append(words[wordIndex]);
      if (wordIndex != words.Count - 1)
      {
        sb.Append(' ');
```

```
        }
      }
      return sb.ToString();
    }
  }
}
```

Фрагмент кода ***ExtensionLib\WordProcessor.cs***

5. Добавьте ссылку на проект `ExtensionLib` в проект `Ch14Ex05`.
6. Модифицируйте код в файле `Program.cs`, как показано ниже:

```
using ExtensionLib;
namespace Ch14Ex05
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Enter a string to convert:");
                        // Ввод строки для преобразования
      string sourceString = Console.ReadLine();
      Console.WriteLine("String with title casing: {0}",
                        // Строка со словами, начинающимися с заглавной буквы
        sourceString.GetWords(capitalizeWords: true)
            .AsSentence());
      Console.WriteLine("String backwards: {0}",
                        // Строка в обратном порядке
        sourceString.GetWords(reverseOrder: true,
          reverseWords: true).AsSentence());
      Console.WriteLine("String length backwards: {0}",
                        // Длина строки в обратном порядке
        sourceString.Length.ToStringReversed());
      Console.ReadKey();
    }
  }
}
```

Фрагмент кода ***Ch14Ex05\Program.cs***

7. Запустите приложение. Когда появится приглашение, введите строку (состоящую хотя бы из 10 символов в длину и более чем одного слова для достижения наилучшего эффекта). На рис. 14.12 показан пример результата, который должен получиться.

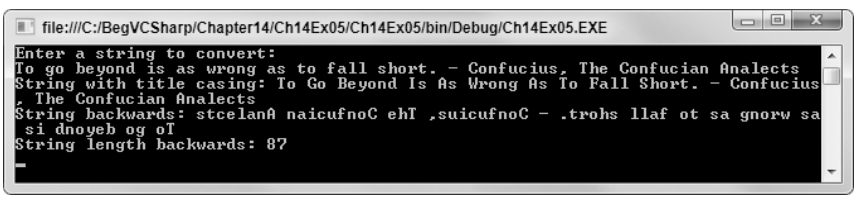


Рис. 14.12*. Приложение* `Ch14Ex05` *в действии*

Описание работы

В этом примере сначала была создана библиотека классов со вспомогательными методами расширения, которая затем использовалась в простом клиентском приложении. Эта библиотека включает расширенную версию статического класса `WordProcessor` из предыдущего практического занятия, которая содержит методы расширения. Пространство

имен ExtensionLib, содержащее этот класс, импортировано в клиентское приложение, что сделало доступными эти методы расширения.

В примере были созданы три метода расширения, описанные в табл. 14.1.

Таблица 14.1. Методы расширения

Метод	Описание
GetWords()	Гибкий метод для манипулирования строкой, как было описано в предыдущем практическом занятии. В этом примере он был изменен так, чтобы стать методом расширения. Возвращает List<string>.
ToStringReversed()	Использует метод Reverse() для обращения порядка букв в строке, возвращаемой после вызова метода ToString() на объекте. Возвращает string.
AsSentence()	Строит на основе объекта List<string> строку, которая состоит из содержащихся внутри него слов.

В клиентском коде каждый из этих методов применялся для изменения вводимой пользователем строки различными способами. Поскольку определенный ранее метод GetWords() возвращает List<string>, его вывод с помощью метода AsSentence() "вытягивается" в строку для облегчения использования.

ToStringReversed() является примером более общего метода расширения. Вместо параметра экземпляра типа string он принимает параметр экземпляра типа object. Это значит, что он может вызываться на *любом* объекте, и он будет появляться в окне IntelliSense для каждого используемого объекта. Делать в нем можно не особо много, поскольку нельзя предполагать, какой объект может использоваться. Можно применить операцию is или попробовать выполнить преобразование, чтобы выяснить, к какому типу относится параметр экземпляра, и в зависимости от этого предпринять соответствующее действе. Либо же, как было сделано в рассмотренном примере, можно использовать базовую функцию, которая поддерживается всеми объектами — метод ToString():

```
public static string ToStringReversed(this object inputObject)
{
  return ReverseWord(inputObject.ToString());
}
```

Этот метод просто вызывает метод ToString() на своем параметре экземпляра и обращает порядок элементов с помощью описанного ранее метода ReverseString(). В примере клиентского приложения метод ToStringReversed() вызывается на переменной int, что приводит к получению строкового представления целого числа цифрами, идущими в обратном порядке.

Методы расширения, которые могут использоваться с множеством типов, очень полезны. Также не забывайте о том, что можно определять обобщенные методы расширения, которые применяют ограничения к используемым типам, как было показано в главе 12.

Лямбда-выражения

Лямбда-выражения (lambda expressions) представляют собой конструкцию, которая появилась в C# 3 и которую можно использовать для упрощения определенных аспектов программирования на языке C#, особенно в сочетании с технологией LINQ. Поначалу освоение лямбда-выражений может вызывать трудности, по большей части из-за того, что они являются очень гибкими в применении. Наиболее полезными они оказываются тогда, когда используются в сочетании с другими функциями языка C#, как, например, с анонимны-

ми методами. Если не принимать во внимание технологию LINQ, которая будет рассматриваться позже в книге, то анонимные методы являются наилучшей отправной точкой для их изучения. Поэтому давайте вкратце вспомним, как работают анонимные методы.

Краткое повторение анонимных методов

Анонимные методы рассматривались в главе 13. Это методы, которые предоставляются внутри кода в тех местах, где в противном случае должна была бы находиться переменная типа делегата. При добавлении обработчика для какого-то события обычно выполняются следующие шаги.

1. Определение метода обработчика событий, возвращаемый тип и параметры которого совпадают с возвращаемым типом и параметрами делегата, закрепленного за интересующим событием.
2. Объявление переменной с типом делегата, используемого для события.
3. Инициализация переменной делегата экземпляром типа делегата, который ссылается на метод обработчика событий.
4. Добавление переменной делегата в список подписчиков события.

На практике дело обстоит немного проще, поскольку заботиться о переменной для хранения делегата обычно никто не хочет и потому при подписке на событие просто используется экземпляр делегата.

Именно так было сделано в следующем коде из главы 13:

```
Timer myTimer = new Timer(100);
myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

В этом коде осуществляется подписка на событие `Elapsed` объекта `Timer`. Это событие использует делегат типа `ElapsedEventHandler`, экземпляр которого создается с помощью идентификатора метода `WriteChar`. В результате при генерации объектом `Timer` события `Elapsed` должен вызываться метод `WriteChar()`. Параметры, передаваемые методу `WriteChar()`, зависят от типов параметров, определенных в делегате `ElapsedEventHandler`, и значений, передаваемых кодом `Timer`, который генерирует событие.

На самом деле, как отмечалось в главе 13, компилятор C# может обеспечить точно такой же результат и в случае предоставления ему меньшего объема кода за счет применения групп методов:

```
myTimer.Elapsed += WriteChar;
```

Компилятору C# известно, делегат какого типа требуется событию `Elapsed`, поэтому он может самостоятельно добавить все недостающие детали. В большинстве случаев, однако, применять такой синтаксис не рекомендуется, поскольку он усложняет восприятие кода и понимание того, что именно в нем происходит. При использовании анонимного метода приведенная выше последовательность шагов сводится к всего лишь одному шагу, который описан ниже.

1. Использование анонимного метода с таким же возвращаемым типом и параметрами, как у делегата, который требуется для подписки на данное событие.

Анонимный метод определяется с помощью ключевого слова `delegate`:

```
myTimer.Elapsed += delegate(object source, ElapsedEventArgs e)
{
  Console.WriteLine (
     "Event handler called after {0} milliseconds.",
     (source as Timer).Interval);
};
```

Этот код работает так же хорошо, как и отдельный обработчик. Главное отличие состоит в том, что применяемый здесь анонимный метод будет фактически скрываться от остальной части кода. То есть, например, использовать этот обработчик повторно в каком-то другом месте внутри приложения нельзя. Кроме того, примененный здесь синтаксис является несколько громоздким. Ключевое слово `delegate` сразу же сбивает с толку, поскольку оно, в сущности, перегружается, т.к. используется и для определения анонимных методов, и для определения типов делегатов.

Использование лямбда-выражений для анонимных методов

Вот мы и подошли, наконец, к использованию лямбда-выраженияй. Лямбда-выражения упрощают синтаксис анонимных методов. На самом деле они позволяют делать не только это, но давайте в этом разделе пока не будем усложнять дело. Используя лямбда-выражения, приведенный выше код можно переписать следующим образом:

```
myTimer.Elapsed += (source, e) => Console.WriteLine(
  "Event handler called after {0} milliseconds.",
  (source as Timer).Interval);
```

На первый взгляд это выражение выглядит несколько устрашающе (кроме тех, кто знаком с так называемыми языками функционального программирования вроде Lisp или Haskell, например). Однако если присмотреться к нему поближе, то можно будет увидеть или, по крайней мере, догадаться о том, каким образом оно работает и какое отношение имеет к анонимному методу, которое собой заменяет. Лямбда-выражение состоит из трех частей:

- список (нетипизированных) параметров в круглых скобках;
- операция =>;
- оператор C#.

Типы параметров выводятся из контекста с применением той же самой логики, которая были описана в разделе “Анонимные типы” ранее в главе. Операция => просто отделяет список параметров от тела выражения, которое выполняется при вызове лямбда-выражения.

Компилятор берет это лямбда-выражение и создает анонимный метод, работающий точно так же, как и приведенный в предыдущем разделе. В действительности он будет компилировать его в тот же или похожий код CIL.

Следующее практическое занятие позволяет посмотреть, что происходит при использовании лямбда-выражений.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Использование простых лямбда-выражений**

1. Создайте новое консольное приложение по имени `Ch14Ex06` и сохраните его в каталоге `C:\BegVCSharp\Chapter14`.
2. Измените код в его файле `Program.cs` следующим образом:

```
namespace Ch14Ex04
{
  delegate int TwoIntegerOperationDelegate(int paramA, int paramB);
  class Program
  {
    static void PerformOperations(TwoIntegerOperationDelegate del)
    {
      for (int paramAVal = 1; paramAVal <= 5; paramAVal++)
      {
```

```
        for (int paramBVal = 1; paramBVal <= 5; paramBVal++)
        {
          int delegateCallResult = del(paramAVal, paramBVal);
          Console.Write("f({0},{1})={2}",
            paramAVal, paramBVal, delegateCallResult);
          if (paramBVal != 5)
          {
             Console.Write(", ");
          }
        }
        Console.WriteLine();
      }
    }
    static void Main(string[] args)
    {
      Console.WriteLine("f(a, b) = a + b:");
      PerformOperations((paramA, paramB) => paramA + paramB);
      Console.WriteLine();
      Console.WriteLine("f(a, b) = a * b:");
      PerformOperations((paramA, paramB) => paramA * paramB);
      Console.WriteLine();
      Console.WriteLine("f(a, b) = (a - b) % b:");
      PerformOperations((paramA, paramB) => (paramA - paramB) % paramB);
      Console.ReadKey();
    }
  }
}
```

Фрагмент кода ***Ch14Ex06\Program.cs***

3. Запустите приложение. На рис. 14.13 показан результат, который должен получиться.

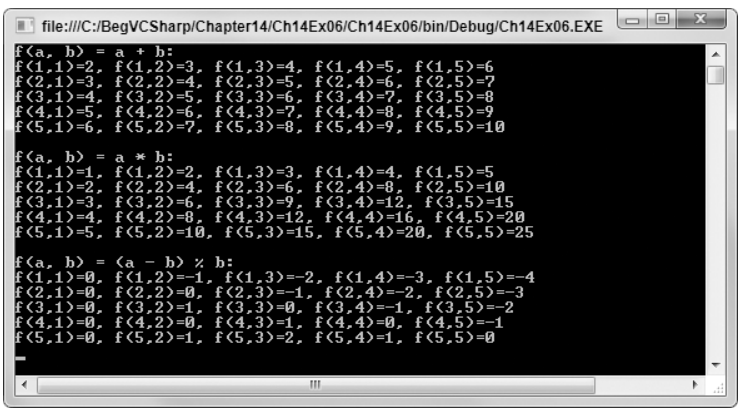


Рис. 14.13*. Приложение* `Ch14Ex06` *в действии*

Описание работы

В этом примере лямбда-выражения применяются для генерации функций, которые могли бы использоваться для возврата результатов выполнения специфической операции над двумя входными параметрами. Затем эти функции обрабатывают 25 пар значений и выводят результаты на консоль.

Первым делом определяется тип делегата с именем TwoIntegerOperationDelegate для представления метода, принимающего два параметра int и возвращающего результат int:

```
delegate int TwoIntegerOperationDelegate(int paramA, int paramB);
```

Он применяется позже при определении лямбда-выражений. Как будет показано ниже, эти лямбда-выражения компилируются в методы, возвращаемый тип и параметры которых соответствуют этому типу делегата.

Затем добавляется метод по имени PerformOperations(), который принимает единственный параметр типа TwoIntegerOperationDelegate:

```
static void PerformOperations(TwoIntegerOperationDelegate del)
{
```

Данному методу можно передать экземпляр делегата (а также анонимный метод или лямбда-выражение, поскольку эти конструкции тоже компилируются в экземпляр делегата), и он будет вызывать метод, который предоставляет этот экземпляр делегата, с рядом значений:

```
  for (int paramAVal = 1; paramAVal <= 5; paramAVal++)
  {
    for (int paramBVal = 1; paramBVal <= 5; paramBVal++)
    {
      int delegateCallResult = del(paramAVal, paramBVal);
```

После этого параметры и результаты выводятся на консоль:

```
      Console.Write("f({0},{1})={2}",
        paramAVal, paramBVal, delegateCallResult);
      if (paramBVal != 5)
      {
        Console.Write(", ");
      }
    }
    Console.WriteLine();
  }
}
```

В методе Main() создаются три лямбда-выражения, которые затем по очереди используются для вызова метода PerformOperations(). Первый из этих вызовов выглядит следующим образом:

```
Console.WriteLine("f(a, b) = a + b:");
PerformOperations((paramA, paramB) => paramA + paramB);
```

Лямбда-выражением здесь является:

```
(paramA, paramB) => paramA + paramB
```

Опять-таки, оно делится на три части.

1. Раздел определения параметров. Здесь определяются два параметра — paramA и paramB. Они не типизированы, а это значит, что компилятор может вывести их типы из контекста. В данном случае он может определить, что вызов метода PerformOperations() требует использования делегата типа TwoIntegerOperationDelegate. У этого типа делегата есть два параметра int, так что за счет выведения компилятор узнает, что paramA и paramB представляют собой переменные типа int.
2. Операция =>. Она просто отделяет параметры от тела лямбда-выражения.
3. Тело выражения. Здесь определяется простая операция, которая предусматривает суммирование параметров paramA и paramB. Обратите внимание — указывать, что это является возвращаемым значением, в ней не требуется. Компилятору известно о том, что

для создания метода, пригодного для использования TwoIntegerOperationDelegate, необходимо, чтобы его возвращаемым типом был int. Поскольку указываемая операция — paramA + paramB — вычисляется в int, и никакой дополнительной информации не предоставлено, компилятор будет считать, что результат данного выражения должен совпадать с возвращаемым типом метода.

В длинном варианте код, использующий это лямбда-выражение, можно развернуть до кода, в котором применяется анонимный метод:

```
Console.WriteLine("f(a, b) = a + b:");
PerformOperations(delegate(int paramA, int paramB)
{
  return paramA + paramB;
});
```

В остальной части кода осуществляется выполнение операций с использованием двух разных лямбда-выражений:

```
Console.WriteLine();
Console.WriteLine("f(a, b) = a * b:");
PerformOperations((paramA, paramB) => paramA * paramB);
Console.WriteLine();
Console.WriteLine("f(a, b) = (a - b) % b:");
PerformOperations((paramA, paramB) => (paramA - paramB) % paramB);
Console.ReadKey();
```

Последнее лямбда-выражение предусматривает выполнение большего объема вычислений, но сложнее из-за этого не становится. Как будет показано далее, синтаксис лямбда-выражений позволяет реализовать и гораздо более сложные операции.

Параметры лямбда-выражений

В коде, который демонстрировался до сих пор, в лямбда-выражениях для определения типов передаваемых параметров применялся механизм выведения типов. На самом деле, это не является обязательным; при желании типы параметров можно определять и явно. Например, можно было бы использовать такое лямбда-выражение:

```
(int paramA, int paramB) => paramA + paramB
```

В этом случае получается более читабельный код, но утрачивается краткость и гибкость. Лямбда-выражения с неявно определенными типами из предыдущего практического занятия можно применять для делегатов с другими числовыми типами, например, long.

Обратите внимание, что использовать явные и неявные типы параметров в одном и том же лямбда-выражении не допускается. Следующее лямбда-выражение, например, компилироваться не будет, потому что для параметра paramA тип был указан явно, а для параметра paramB — неявно:

```
(int paramA, paramB) => paramA + paramB
```

Списки параметров в лямбда-выражениях всегда представляют собой разделенные запятыми перечни параметров, причем все параметры типизированы либо явно, либо неявно. При наличии только одного параметра круглые скобки можно опускать; в противном случае они являются обязательной частью списка параметров. Например, вот как выглядит лямбда-выражение с одним неявно типизированным параметром:

```
param1 => param1 * param1
```

Можно также определять лямбда-выражения, не имеющие параметров. Для этого указываются пустые круглые скобки:

```
() => Math.PI
```

Такое выражение можно использовать там, где требуется делегат, не принимающий параметров и возвращающий значение double.

Тело операторов лямбда-выражений

Во всем приведенном до сих пор коде в теле оператора лямбда-выражений использовалось единственное выражение. Это выражение интерпретировалось как возвращаемое значение лямбда-выражения, что, например, показывает, каким образом выражение paramA + paramB применять в качестве тела лямбда-выражения, предназначенного для делегата с возвращаемым типом int (при условии явного или неявного приведения обоих параметров paramA и paramB к int, как это было сделано в примере).

В одном из предыдущих примеров было показано, как делегат с возвращаемым типом void позволяет использовать менее громоздкий код в теле оператора:

```
myTimer.Elapsed += (source, e) => Console.WriteLine(
  "Event handler called after {0} milliseconds.",
  (source as Timer).Interval);
```

Здесь оператор ни во что не вычисляется, и потому будет выполняться без использования возвращаемого значения.

Из-за того, что лямбда-выражения могут быть представлены как расширение синтаксиса анонимных методов, вряд ли покажется удивительным, что в теле лямбда-выражений допускается также применять множество операторов. Для этого необходимо просто написать заключенный в фигурные скобки блок кода, во многом подобно тому, как это делается в любых других ситуациях, когда возникает необходимость предоставлять несколько строк кода:

```
(param1, param2) =>
{
  // Несколько операторов.
}
```

В случае использования лямбда-выражения в сочетании с делегатом, возвращаемым типом которого является не void, нужно обязательно выполнять возврат значения с помощью ключевого слова return, точно так же как и в любом другом методе:

```
(param1, param2) =>
{
  // Несколько операторов.
  return returnValue ;
}
```

Например, ранее уже показывалось, что код из последнего практического занятия:

```
PerformOperations((paramA, paramB) => paramA + paramB);
```

можно переписать следующим образом:

```
PerformOperations(delegate(int paramA, int paramB)
{
  return paramA + paramB;
});
```

В качестве альтернативы этот код можно переписать и так:

```
PerformOperations((paramA, paramB) =>
{
  return paramA + paramB;
});
```

В этом случае сохраняется большая преемственность с исходным кодом, поскольку остается неявная типизация параметров paramA и paramB.

По большей части лямбда-выражения оказываются наиболее полезными и, конечно же, наиболее элегантными тогда, когда применяются в виде одиночных выражений. Честно говоря, когда требуется несколько операторов, код получится более удобным для чтения и многократного использования, если будет определен в виде отдельного неанонимного метода, а не лямбда-выражения.

Лямбда-выражения как делегаты и деревья выражений

Ранее уже были отмечены некоторые отличия между лямбда-выражениями и анонимными методами, такие как более высокая гибкость лямбда-выражений (например, возможность использования в них неявно типизированных параметров). Теперь пора рассмотреть еще одно важное отличие, последствия которого станут более понятны позже в настоящей книге, при рассмотрении технологии LINQ.

Лямбда-выражения можно интерпретировать двумя способами. Первый способ, который демонстрировался повсюду в этой главе, позволяет интерпретировать их в качестве делегата, т.е. присваивать лямбда-выражение переменной типа делегата, как это делалось в предыдущем практическом занятии.

В общих чертах это означает, что лямбда-выражение с параметрами в количестве до восьми можно представлять в виде одного из следующих обобщенных типов, определения которых находятся в пространстве имен `System`.

- `Action` — для представления лямбда-выражений, не имеющих параметров и возвращающих тип `void`.
- `Action<>` — для представления лямбда-выражений, имеющих параметры в количестве до восьми и возвращающих тип `void`.
- `Func<>` — для представления лямбда-выражений, имеющих параметры в количестве до восьми, но возвращающих тип, отличный от `void`.

В `Action<>` допускается указывать до восьми параметров обобщенного типа, по одному для каждого параметра, а в `Func<>` — до девяти параметров обобщенного типа, т.е. восемь для параметров и один для возвращаемого типа. В `Func<>` возвращаемый тип должен быть указан последним в списке.

Например, следующее лямбда-выражение (которое уже приводилось ранее):

```
(int paramA, int paramB) => paramA + paramB
```

можно было бы представить в виде делегата типа `Func<int, int, int>`, поскольку оно имеет два параметра и возвращаемый тип, причем все они относятся к типу `int`.

Второй способ позволяет интерпретировать лямбда-выражение как *дерево выражения* (expression tree). Под деревом выражения понимается абстрактное представление лямбда-выражения, и потому выполнять его напрямую нельзя. Вместо этого его можно использовать для анализа лямбда-выражения программным образом и выполнения в ответ на него различных действий.

Очевидно, что это сложная тема. Тем не менее, деревья выражений играют критически важную роль в функциональности LINQ. Чтобы привести более конкретный пример, библиотека LINQ содержит обобщенный класс по имени `Expression<>`, который служит для инкапсуляции лямбда-выражения. Один из способов использования этого класса состоит в том, чтобы взять лямбда-выражение, которое было написано разработчиком C#, и преобразовать его в эквивалентный сценарий на языке SQL, пригодный для выполнения непосредственно в базе данных.

На данном этапе пока больше ничего знать не требуется. Когда данная функциональность встретится далее в книге, будут предоставлены дополнительные пояснения.

Лямбда-выражения и коллекции

Теперь, когда известно об обобщенном делегате `Func<>`, можно переходить к рассмотрению других методов расширения, которые предоставляются пространством имен `System.Linq` для типов-массивов (и которые, возможно, приходилось встречать в IntelliSense на различных этапах написания кода). Например, существует метод расширения `Aggregate()`, который имеет три следующих перегруженных версии:

```
public static TSource Aggregate<TSource>(
 this IEnumerable<TSource> source,
  Func<TSource, TSource, TSource> func);

public static TAccumulate Aggregate<TSource, TAccumulate>(
  this IEnumerable<TSource> source,
  TAccumulate seed,
  Func<TAccumulate, TSource, TAccumulate> func);

public static TResult Aggregate<TSource, TAccumulate, TResult>(
  this IEnumerable<TSource> source,
  TAccumulate seed,
  Func<TAccumulate, TSource, TAccumulate> func,
  Func<TAccumulate, TResult> resultSelector);
```

Как и показанный ранее метод расширения, этот метод на первый взгляд выглядит непонятно, но если разбить его на части, то все в нем станет довольно просто. В окне IntelliSense описание предназначения этого метода будет выглядеть так:

```
Applies an accumulator function over a sequence.
```

Применяет аккумулирующую функцию к последовательности.

Это означает, что аккумулирующая функция (которая может предоставляться в виде лямбда-выражения) будет применяться для каждой пары элементов в коллекции от начала до конца с превращением выходных данных каждой операции вычисления во входные данные следующей операции вычисления.

Самая простая из трех перегруженных версий предусматривает указание только одного обобщенного типа, который может выводиться из типа параметра экземпляра. Например, в следующем коде этим обобщенным типом будет `int` (аккумулирующая функция пока оставлена пустой):

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate(...);
```

Этот код эквивалентен следующему:

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate < int > (...);
```

Требуемое лямбда-выражение здесь может быть выведено из спецификации метода расширения. Поскольку типом `TSource` в этом коде является `int`, лямбда-выражение должно представлять собой делегат вида `Func<int, int, int>`. Например, это может быть и такое лямбда-выражение, как показанное ранее:

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate((paramA, paramB) => paramA + paramB);
```

При таком вызове лямбда-выражение будет вызываться дважды: первый раз — с параметром `paramA` равным 2 и параметром `paramB` равным 6, а второй — с параметром `paramA` равным 8 (полученным в результате первого вычисления) и параметром `paramB` равным 3. В конечном результате переменной `result` будет присваиваться `int`-значение `11`, представляющее собой сумму значений всех элементов в массиве.

Две других перегруженных версии метода расширения `Aggregate()` выглядят похоже, но позволяют выполнять немного более сложные операции по обработке и потому более детально иллюстрируются в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование лямбда-выражений с коллекциями

1. Создайте новое консольное приложение по имени `Ch14Ex07` и сохраните его в каталоге `C:\BegVCSharp\Chapter14`.

2. Измените код в его файле `Program.cs` следующим образом:

```
static void Main(string[] args)
{
  string[] curries = { "pathia", "jalfreze", "korma" };
  Console.WriteLine(curries.Aggregate(
    (a, b) => a + " " + b));
  Console.WriteLine(curries.Aggregate < string, int > (
    0,
   (a, b) => a + b.Length));
  Console.WriteLine(curries.Aggregate < string, string, string > (
   "Some curries:",
    (a, b) => a + " " + b,
    a => a));
  Console.WriteLine(curries.Aggregate < string, string, int > (
    "Some curries:",
    (a, b) => a + " " + b,
    a => a.Length));
  Console.ReadKey();
}
```

Фрагмент кода ***Ch14Ex07\Program.cs***

3. Запустите приложение. На рис. 14.14 показан результат, который должен получиться.

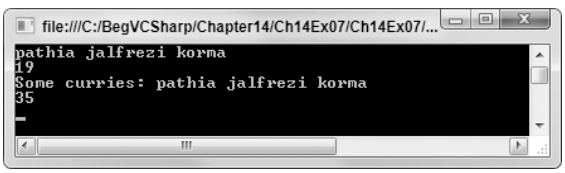


Рис. 14.14*. Приложение* `Ch14Ex07` *в действии*

Описание работы

В этом примере проводились эксперименты со всеми тремя перегруженными версиями метода расширения `Aggregate()` с использованием в качестве исходных данных строкового массива, содержащего три элемента.

Сначала выполняется простая конкатенация:

```
Console.WriteLine(curries.Aggregate(
  (a, b) => a + " " + b));
```

Первая пара элементов с помощью простого синтаксиса и конкатенации превращается в одну строку. Такой синтаксис является далеко не наилучшим для выполнения конкатенации строк — в идеале вместо него для достижения оптимальной производительности было бы лучше использовать либо метод `string.Concat()`, либо метод `string.Format()` — но здесь он позволяет легко увидеть, что происходит. После этой первой операции конкатенации результат передается обратно лямбда-выражению вместе с третьим элементом массива

во многом подобно тому, как это происходило при суммировании значений int, которое показывалось ранее. Это в конечном итоге приводит к выполнению конкатенации всего массива с разделением его элементов пробелами.

Далее применяется вторая перегруженная версия функции Aggregate(), которая имеет два параметра обобщенного типа TSource и TAccumulate. В этом случае лямбда-выражение должно иметь вид Func<TAccumulate, TSource, TAccumulate>. Кроме того, необходимо указать начальное значение для типа TAccumulate. Это начальное значение используется в первом вызове лямбда-выражения вместе с первым элементом массива. В последующих вызовах берется аккумулированный результат предыдущих вызовов этого выражения. Код, который использовался в данном примере, выглядит следующим образом:

```
Console.WriteLine(curries.Aggregate<string, int>(
  0,
  (a, b) => a + b.Length));
```

Здесь типом аккумулятора (и, косвенно, возвращаемого значения) является int. Значение аккумулятора устанавливается в исходное значение 0 и с каждым вызовом лямбда-выражения суммируется с длиной соответствующего элемента массива. В конечном счете, получается сумма значений длины всех элементов в массиве.

Далее идет последняя перегруженная версия метода Aggregate(), которая тоже предусматривает указание трех параметров обобщенного типа и отличается от предыдущей версии только тем, что тип возвращаемого значения в ней может не совпадать ни с типом элементов в массиве, ни с типом значения аккумулятора. Первым делом эта перегруженная версия применяется для конкатенации строковых элементов с начальной строкой:

```
Console.WriteLine(curries.Aggregate<string, string, string>(
  "Some curries:",
  (a, b) => a + " " + b,
  a => a));
```

Последний параметр в этом методе, resultSelector, должен быть обязательно указан даже в том случае, если (как и в рассматриваемом примере) значение аккумулятора просто копируется в результат. Этот параметр представляет собой лямбда-выражение типа Func<TAccumulate, TResult>.

И, наконец, в последнем разделе кода та же самая версия Aggregate() используется снова, но на этот раз с возвращаемым значением int. Здесь параметр resultSelector предоставляется с лямбда-выражением, которое возвращает длину строки аккумулятора:

```
Console.WriteLine(curries.Aggregate<string, string, int>(
  "Some curries:",
  (a, b) => a + " " + b,
  a => a.Length));
```

Ничего особого зрелищного в этом примере нет, но зато он действительно демонстрирует применение более сложных методов расширения, которые имеют параметры обобщенного типа, коллекции, и, очевидно, обладают более сложным синтаксисом. Такие методы будут неоднократно встречаться далее в этой книге.

Резюме

В настоящей главе рассматривались новые и недавно добавленные средства версии C# 4, которая поставляется в составе Visual Studio 2010 и Visual C# Express Edition 2010. Было показано, как они упрощают кодирование ряда часто используемой и/или более сложной функциональности.

В главе были освещены следующие вопросы.

- Использование инициализаторов объектов и коллекций для создания и инициализации объектов и коллекций за один шаг.

- Выведение типов из контекста IDE-средой и компилятором C# и применение ключевого слова `var`.
- Создание и использование анонимных типов, которые комбинируют инициализаторы и механизм выведения типов.
- Применение технологии динамического просмотра к переменным, которая обеспечивает запрашивание их членов только во время выполнения.
- Использование именованных и необязательных параметров для более гибкого вызова методов.
- Создание методов расширения, которые можно вызывать на экземплярах других типов, без добавления кода в определение этих типов, а также применение этого приема для построения библиотек вспомогательных методов.
- Использование лямбда-выражений для предоставления анонимных методов экземплярам делегатов и применение расширенного синтаксиса лямбда-выражений для получения дополнительной функциональности.

Большинство рассмотренных в этой главе средств было добавлено в C# специально для применения вместе с такой новой технологией .NET Framework, как LINQ. Степень элегантности и многие тонкие детали показанного здесь кода станут более очевидны позже. Несмотря на это, некоторые из продемонстрированных здесь приемов являются настолько мощными, что их можно применять уже прямо сейчас, совершенствуя свои навыки программирования на C#.

На этом рассмотрение языка C# в том виде, в котором он представлен в версии 4, завершено. Однако для программирования с использованием .NET Framework одних только знаний C# недостаточно. Язык C# обеспечивает лишь средствами, которые необходимы для написания .NET-приложений, а классы, доступные в .NET Framework, предоставляют тот исходный материал, с которого необходимо начинать разработку. Далее в книге основное внимание будет уделяться именно этим классам и решению с их помощью разнообразных задач. В следующей главе будет использоваться развитая функциональность Windows Forms для создания графических пользовательских интерфейсов. Навыки, приобретенные в первой части книги, пригодятся при проработке всех последующих глав.

Упражнения

1. Модифицируйте следующий класс так, чтобы стало возможным использование инициализатора объектов, и приведите пример кода для создания и инициализации экземпляра этого класса за один шаг.

```
public class Giraffe
{
  public Giraffe(double neckLength, string name)
  {
    NeckLength = neckLength;
    Name = name;
  }
  public double NeckLength {get; set;}
  public string Name {get; set;}
}
```

2. Верно ли то, что в случае объявления переменной типа `var` ее можно будет использовать для хранения объектов любого типа?
3. Как в случае использования анонимных типов можно сравнивать два экземпляра, чтобы выяснить, содержатся ли в них одинаковые данные?

4. Попробуйте исправить код следующего метода расширения, в котором присутствует ошибка:

```
public string ToAcronym(this string inputString)
{
  inputString = inputString.Trim();
  if (inputString == "")
  {
    return "";
  }
  string[] inputStringAsArray = inputString.Split(' ');
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < inputStringAsArray.Length; i++)
  {
   if (inputStringAsArray[i].Length > 0)
   {
     sb.AppendFormat("{0}", inputStringAsArray[i].Substring(0, 1).ToUpper());
   }
  }
  return sb.ToString();
}
```

5. Как сделать так, чтобы метод расширения в предыдущем упражнении был доступен клиентскому коду?

6. Перепишите показанный выше метод `ToAcronym()` так, чтобы его код занимал одну строку, и позаботьтесь о том, чтобы строки с множеством пробелов между словами в нем не приводили к появлению ошибок. Подсказка: для этого потребуется использовать тернарную операцию `?:`, функцию расширения `string.Aggregate<string, string>()` и лямбда-выражение.

Решения упражнений приведены в приложении A.

Что вы узнали в этой главе

Тема	Основные концепции
Инициализаторы	Инициализаторы можно использовать для инициализации объектов и коллекций во время их создания. Оба типа инициализаторов состоят из блока кода в фигурных скобках. Инициализаторы объектов позволяют устанавливать значения для свойств за счет предоставления списка пар “имя/значение”, разделенных запятыми. Инициализаторы коллекций представляют собой списки значений, разделенных запятыми. Инициализаторы объектов можно использовать вместе с конструкторами не по умолчанию.
Выведение типов	Ключевое слово `var` позволяет не указывать тип переменной во время ее объявления. Однако это возможно, только если этот тип может быть определен на этапе компиляции. Использование `var` не нарушает строгой типизации в C#, поскольку переменная, объявленная с помощью `var`, в конечном счете, получает один и только один возможный тип.
Анонимные типы	Для многих простых типов, применяемых для структурирования хранилища данных, явное определение типа не обязательно. Члены анонимного типа выводятся на основе их использования. Анонимный тип определяется с помощью синтаксиса объектного инициализатора, и каждое устанавливаемое свойство определяется как свойство только для чтения.

Тема	Основные концепции
Динамический просмотр	Ключевое слово `dynamic` служит для определения переменной динамического типа, которая может содержать любое значение. После этого во время выполнения можно получать доступ к членам этого значения, используя обычный синтаксис свойств или методов. При попытке обратиться во время выполнения к несуществующему члену возникает исключение. Такая динамическая типизация значительно упрощает синтаксис, который необходим для доступа к типам, отличным от .NET, или к типам .NET, чья информация о типах не доступна на этапе компиляции. Однако динамические типы должны применяться осторожно, поскольку при этом утрачивается проверка кода во время компиляции. Поведением динамического просмотра можно управлять, реализовав интерфейс `IDynamicMetaObjectProvider`.
Необязательные параметры методов	Часто приходится иметь дело с методами, объявленными с большим числом параметров, многие из которых используются редко. Вместо того чтобы принуждать указывать в клиентском коде значения для таких параметров, можно предоставить набор перегрузок метода. В качестве альтернативы, редко используемые параметры можно определить как необязательные (и предусмотреть для них значения по умолчанию на случай, если они не будут заданы при вызове). В таком случае в клиентском коде можно будет указывать столько параметров, сколько необходимо.
Именованные параметры методов	Указывать параметры в клиентском коде можно позиционно или по именам (или обоими способами, но позиционные параметры должны быть первыми). Именованные параметры допускается передавать в любом порядке. Это полезно в комбинации с необязательными параметрами.
Методы расширения	Методы расширения можно определять для любого существующего типа, не модифицируя его определение. Это касается также и системных типов, таких как `string`. Методы расширения определяются как статические методы необобщенного статического класса. Первый параметр такого метода определяется с ключевым словом `this` и представляет собой значение экземпляра, для которого данный метод вызывается. После определения статический метод может вызываться в любом коде, имеющем ссылку на пространство имен, которое содержит класс, определяющий этот метод. Методы расширения могут вызываться из экземпляров типа, используемого в определении метода, или любого унаследованного типа. Это позволяет определять универсальные методы расширения для семейств типов. Другим способом создания универсальных методов расширения является их определение для использования с отдельным интерфейсом.
Лямбда-выражения	Лямбда-выражения — это, в сущности, сокращенный способ определения анонимных методов, хотя они обладают дополнительными возможностями, такими как неявная типизация. Лямбда-выражение определяется в виде списка разделенных запятыми параметров (или пары скобок, если параметров нет), за которым следует операция `=>` и выражение. Выражением может быть блок кода, заключенный в фигурные скобки. Лямбда-выражения могут принимать до восьми параметров и имеют необязательный возвращаемый тип. Они могут быть представлены с помощью типов делегатов `Action`, `Action<>` и `Func<>`. Многие методы расширения LINQ, которые могут применяться с коллекциями, используют параметры в виде лямбда-выражений.

ЧАСТЬ II

Программирование Windows-приложений

В ЭТОЙ ЧАСТИ...

15

Основы программирования для Windows

В ЭТОЙ ГЛАВЕ...

- Визуальный конструктор Windows Forms
- Элементы управления, предназначенные для отображения информации пользователю, такие как `Label` и `LinkLabel`
- Элементы управления, предназначенные для запуска событий, такие как `Button`
- Элементы управления, которые позволяют пользователю вводить текст, такие как `TextBox`
- Элементы управления, которые позволяют информировать пользователя о текущем состоянии приложения и изменять это состояние, такие как `RadioButton` и `CheckButton`
- Элементы управления, позволяющие отображать информацию в виде списков, такие как `ListBox` и `ListView`
- Элементы управления, позволяющие группировать другие элементы управления, такие как `TabControl` и `Groupbox`

Около 10 лет назад Visual Basic был встречен с огромным энтузиазмом, поскольку он предложил программистам инструментальные средства создания чрезвычайно детализированных интерфейсов пользователя с помощью интуитивно понятного визуального конструктора форм и простой для изучения язык программирования. В сочетании друг с другом они образуют, пожалуй, наилучшую на сегодняшний день среду для быстрой разработки приложений (Rapid Application Development — RAD). Одно из преимуществ, предоставляемых такими средствами RAD, как Visual Basic, состоит в том, что они обеспечивают доступ к ряду заранее созданных элементов управления, которые можно использовать для быстрого построения интерфейса пользователя приложения.

В основе разработки большинства приложений Visual Basic для Windows лежит применение средств визуального конструктора форм (Forms Designer). Создание интерфейса пользователя осуществляется перетаскиванием элементов управления из панели инструментов на форму и их размещением там, где они должны отображаться во время выполнения программы. При этом двойной щелчок на элементе управления добавляет дескриптор данного элемента управления. Элементы управления, предлагаемые Microsoft, а также дополнительные нестандартные элементы управления, которые можно приобрести по умеренным ценам, снабдили программистов невероятно обширным арсеналом многократно используемого, тщательно протестированного кода, доступного единственным щелчком кнопкой мыши. Теперь такой подход к построению приложений доступен и разработчикам на C# через Visual Studio.

В этой главе мы поработаем с Windows Forms и воспользуемся некоторыми из множества элементов управления, поставляемых с Visual Studio. Эти элементы управления охватывают широкий диапазон функциональности, и благодаря средствам конструирования Visual Studio, разработка интерфейсов пользователей и реализация взаимодействия с пользователем становится очень простой задачей. Рассмотрение всех элементов управления Visual Studio в рамках этой книги невозможно, потому в настоящей главе описаны только некоторые из наиболее часто применяемых компонентов, начиная с меток и текстовых полей и заканчивая списочными представлениями и элементами управления с вкладками.

Элементы управления

Хоть это может быть и не заметно, но при работе с Windows Forms вы имеете дело с пространством имен `System.Windows.Forms`. Это пространство имен указано в директивах `using` в одном из файлов, содержащих класс `Form`. Большинство элементов управления в .NET являются производными от класса `System.Windows.Forms.Control`. Этот класс определяет базовую функциональность элементов управления, вследствие чего многие свойства и события элементов управления, с которыми придется встретиться, идентичны. Многие из этих классов сами являются базовыми для других элементов управления, как это имеет место в случае с классами `Label` и `TextBoxBase` (рис. 15.1).

Свойства

Все элементы управления имеют ряд свойств, которые служат для манипулирования их поведением. Базовый класс большинства элементов управления, `System.Windows.Forms.Control`, обладает набором свойств, которые другие элементы управления наследуют непосредственно или переопределяют для обеспечения того или иного нестандартного поведения.

Некоторые из наиболее часто используемых свойств класса `Control` описаны в табл. 15.1. Эти свойства представлены в большинстве элементов управления, рассмотренных в данной главе, потому мы не будем повторно подробно останавливаться на них, если только поведение рассматриваемого элемента управления не потребует изменений. Приведенная таблица не является исчерпывающей. Если хотите ознакомиться со всеми свойствами этого класса, обратитесь к документации .NET Framework SDK.

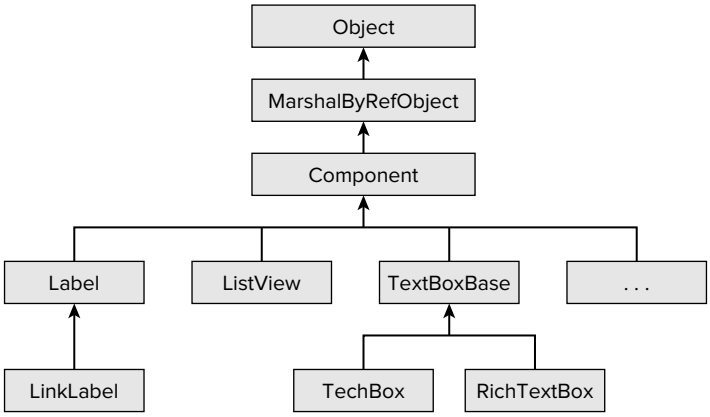


Рис. 15.1. Иерархия классов для элементов управления

Таблица 15.1. Часто используемые свойства класса `Control`

Свойство	Описание
`Anchor`	Указывает поведение элемента управления при изменении размеров его контейнера. Это свойство подробно рассматривается в следующем разделе
`BackColor`	Цвет фона элемента управления
`Bottom`	Указывает расстояние от верхнего края окна до нижнего края элемента управления. Это не то же самое, что и высота элемента управления
`Dock`	Стыкует элемент управления к краям его контейнера. Это свойство более подробно описано в следующем разделе
`Enabled`	Установка значения свойства `Enabled` равным `true` означает, что элемент управления может принимать данные, вводимые пользователем. Установка этого значения в `false` означает невозможность приема этих данных
`ForeColor`	Цвет переднего плана элемента управления
`Height`	Расстояние между верхним и нижним краями элемента управления
`Left`	Положение левого края элемента управления относительно левого края его контейнера
`Имя`	Имя элемента управления. Это имя может использоваться для ссылки на элемент управления в коде
`Parent`	Родительский элемент для данного элемента управления
`Right`	Положение правого края элемента управления относительно левого края его контейнера
`TabIndex`	Порядковый номер элемента управления в последовательности обхода элементов внутри контейнера по клавише <Tab>
`TabStop`	Указывает доступность элемента управления по клавише <Tab>
`Tag`	Обычно это значение не используется самим элементом управления. Оно позволяет хранить информацию об элементе управления в самом элементе управления. В визуальном конструкторе форм этому свойству можно присваивать только строку

Окончание табл. 15.1

Свойство	Описание
`Text`	Содержит текст, связанный с данным элементом управления
`Top`	Положение верхнего края элемента управления относительно верхнего края его контейнера
`Visible`	Указывает видимость элемента управления во время выполнения
`Width`	Ширина элемента управления

Привязка, стыковка и определение формы элементов управления

Еще в версии Visual Studio 2005 используемая по умолчанию рабочая область визуального конструктора форм изменилась с сеточной поверхности, на которой можно было размещать элементы управления, на гладкую поверхность, использующую линии привязки для позиционирования элементов управления. Переключаться между этими двумя стилями конструирования можно с помощью пункта Options (Параметры) меню Tools (Сервис). Выберите узел Windows Forms Designer (Визуальный конструктор форм) в дереве слева и установите параметр Layout Mode (Режим макета). Выбор наиболее подходящего инструментального средства в значительной мере зависит от личных предпочтений. В следующем практическом занятии используется режим по умолчанию.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование линий привязки

Чтобы попрактиковаться в применении линий привязки в Windows Forms Designer, выполните перечисленные ниже действия.

1. Создайте новое приложение Windows Forms и назовите его `SnapLines`.
2. Перетащите элемент кнопки (`Button`) из панели управления Toolbox на середину формы.
3. Перетащите кнопку в верхний левый угол формы. Обратите внимание, что при приближении элемента управления к краю формы появляются линии, исходящие от ее левого и верхнего краев, а элемент управления помещается в фиксированную позицию. Элемент управления можно переместить за пределы линий привязки или оставить в данной позиции.
4. Снова переместите кнопку в центр формы и перетащите на форму еще одну кнопку из Toolbox. Переместите ее под существующую кнопку. При перемещении кнопки под существующую появляются линии привязки. Эти линии привязки позволяют выравнивать элементы управления так, чтобы они размещались непосредственно над или на одной высоте с другим элементом управления. При перемещении новой кнопки по направлению к существующей другие линии привязки позволяют размещать кнопки на заблаговременно определенном расстоянии между ними.
5. Измените размер кнопки `button1`, чтобы она была шире, чем кнопка `button2`. Затем измените также размеры кнопки `button2` и обратите внимание, что когда ее ширина становится равной ширине кнопки `button1`, появляются линии привязки, которые позволяют установить ширину элементов управления одинаковой.
6. Добавьте в форму элемент управления `TextBox`, поместив его под кнопками, и измените его свойство `Text` на `Hello World!`

7. Добавьте на форму элемент управления `Label` и поместите его слева от элемента `TextBox`. Обратите внимание, что при перемещении элемента управления появляются линии привязки, которые позволяют привязать его к верхнему и нижнему краю элемента управления `TextBox`, но между ними появляется также третья линия привязки. Эта линия привязки позволяет разместить элемент `Label` так, чтобы текст в элементах управления `TextBox` и `Label` отображался на одной высоте.

Свойства `Anchor` и `Dock`

Свойства `Anchor` и `Dock` особенно полезны при конструировании формы. Задача предотвращения искажения формы при изменении размеров окна пользователем далеко не тривиальна, и раньше для этого требовалось написание множества строк кода. Во многих программах эта проблема решается просто путем запрещения изменений размеров окна. Этот способ решения проблемы является простейшим, но не всегда наилучшим. Свойства `Anchor` и `Dock`, появившиеся в .NET, позволяют решить эту проблему, не написав ни единой строки кода.

Свойство `Anchor` задает поведение элемента управления при изменении размеров окна пользователем. Можно указать, чтобы элемент управления изменял свои размеры, сохраняя пропорции своих краев, либо сохранял свои размеры, сберегая позицию относительно краев окна.

Свойство `Dock` указывает, что элемент управления должен пристыковываться к краю своего контейнера. Если пользователь изменяет размеры окна, элемент управления остается пристыкованным к краю окна. Если, например, указать, что элемент управления должен быть пристыкован к нижнему краю своего контейнера, то элемент управления изменяет свои размеры и/или перемещается, чтобы всегда занимать нижнюю часть окна независимо от своих размеров.

Более подробно свойство `Anchor` описано далее в этой главе.

События

В главе 13 вы узнали, что собой представляют события, и как они используются. В этом разделе освещены конкретные виды событий — а именно, те, которые генерируются элементами управления Windows Forms. Обычно эти события связаны с действиями, выполняемыми пользователем. Например, когда пользователь щелкает на кнопке, кнопка генерирует событие, указывающее, что с ней произошло. Обработка события — это средство, посредством которого программист может снабдить данную кнопку той или иной функциональностью.

Класс `Control` определяет набор событий, которые присущи всем элементам управления, используемым в этой главе. Некоторые из них описаны в табл. 15.2. Как и в предыдущей таблице, здесь приведена всего лишь подборка наиболее часто используемых событий. Если хотите ознакомиться с полным перечнем событий, обратитесь к документации .NET Framework SDK.

Многие из этих событий используются в примерах этой главы. Все примеры следуют одному и тому же формату: вначале мы создаем визуальное представление формы, выбирая и размещая элементы управления, и лишь затем приступаем добавлению обработчиков событий — основной объем работы выполняется именно на этом этапе.

Существуют три основных способа обработки конкретного события. Первый — двойной щелчок на элементе управления. В результате происходит обращение к обработчику события, используемого по умолчанию данным элементом управления — это событие отличается в зависимости от элемента управления. Если это событие то, что требуется — прекрасно. В противном случае доступны два возможных варианта.

Один — использование списка событий в окне Properties (Свойства), показанного на рис. 15.2, который отображается при щелчке на кнопке со значком молнии.

Таблица 15.2. Часто используемые события класса `Control`

Событие	Описание
`Click`	Происходит при щелчке на элементе управления. В некоторых случаях это событие происходит также при нажатии пользователем клавиши <Enter>
`DoubleClick`	Происходит при двойном щелчке на элементе управления. Обработка события `Click` для некоторых элементов управления, таких как `Button`, полностью исключает возможность вызова события `DoubleClick`
`DragDrop`	Происходит по завершении операции перетаскивания и оставления — иначе говоря, при перетаскивании объекта поверх элемента управления и освобождении кнопки мыши пользователем
`DragEnter`	Происходит, когда перетаскиваемый объект перемещается внутрь границ элемента управления
`DragLeave`	Происходит, когда перетаскиваемый объект покидает границы элемента управления
`DragOver`	Происходит, когда объект перетаскивается поверх элемента управления
`KeyDown`	Происходит при нажатии клавиши, пока элемент управления находится в фокусе. Это событие всегда происходит прежде событий `KeyPress` и `KeyUp`
`KeyPress`	Происходит при нажатии клавиши, в то время как элемент управления находится в фокусе. Это событие всегда происходит после события `KeyDown` и перед событием `KeyUp`. Различие между событиями `KeyDown` и `KeyPress` состоит в том, что событие `KeyDown` передает код нажатой клавиши, а событие `KeyPress` передает соответствующее значение `char` клавиши
`KeyUp`	Происходит при освобождении клавиши, в то время как элемент управления находится в фокусе. Это событие всегда происходит после событий `KeyDown` и `KeyPress`
`GotFocus`	Происходит, когда элемент управления получает фокус. Это событие не следует использовать для выполнения проверки достоверности. Вместо него должны применяться события `Validating` и `Validated`
`LostFocus`	Происходит, когда элемент управления теряет фокус. Это событие не следует использовать для выполнения проверки достоверности. Вместо него должны применяться события `Validating` и `Validated`
`MouseDown`	Происходит при помещении указателя мыши над элементом управления и нажатии кнопки мыши. Это событие не эквивалентно событию `Click`, поскольку `MouseDown` происходит сразу после нажатия кнопки мыши и перед ее освобождением
`MouseMove`	Происходит непрерывно в процессе перемещения указателя мыши над элементом управления
`MouseUp`	Происходит при помещении указателя мыши над элементом управления и освобождении кнопки мыши
`Paint`	Происходит при прорисовке элемента управления
`Validated`	Запускается, когда элемент управления, свойство `CausesValidation` которого установлено в `true`, готов принять фокус. Это событие запускается по завершении события `Validating` и оно указывает на завершение проверки
`Validating`	Запускается, когда элемент управления, свойство `CausesValidation` которого установлено в `true`, готов принять фокус. Обратите внимание, что проверяемым элементом управления является тот, который теряет фокус, а не тот, который его получает

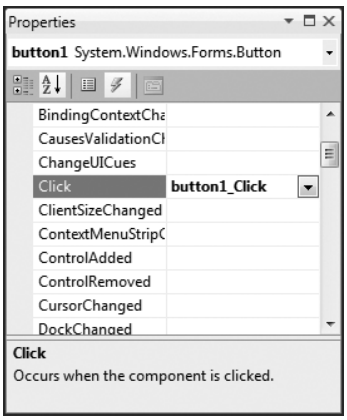


Рис. 15.2. *Работа с событиями в окне* Properties

Чтобы добавить обработчик для конкретного события, дважды щелкните на этом событии в списке событий. В результате будет сгенерирован код подписки элемента управления на событие и сигнатура метода обработки этого события. Или же можно ввести имя метода для обработки конкретного события в поле, расположенном рядом с данным событием в списке событий, и после нажатия клавиши <Enter> будет сгенерирован обработчик события с выбранным именем.

Вторая возможность — добавление кода подписки на событие вручную. Даже при вводе кода, необходимого для подписки на событие, Visual Studio обнаруживает выполняемое действие и предлагает добавить в код сигнатуру метода, как если бы операция выполнялась из Forms Designer.

Каждый из этих двух подходов требует выполнения двух действий — подписки на событие (т.е. связывания с ним) и создания соответствующей сигнатуры метода обработчика. Двойной щелчок на элементе управления и попытка обработки другого события посредством замены сигнатуры метода события, используемого по умолчанию, на событие, которое действительно нужно обработать, окажется безуспешной — необходимо также изменить код подписки на событие в методе `InitializeComponent()`. Поэтому упомянутый обходной путь в действительности не может служить быстрым способом обработки конкретных событий.

Теперь можно приступать к рассмотрению самих элементов управления. Мы начнем с того, с который, без сомнения, приходится использовать несчетное количество раз при работе с Windows-приложениями: элемента управления `Button`.

Элемент управления `Button`

Когда говорят о кнопке, вероятно, в первую очередь на ум приходит прямоугольная кнопка, на которой можно щелкать для выполнения той или иной задачи. Однако .NET Framework предоставляет класс, производный от класса `Control` — `System.Windows.Forms.ButtonBase`, — который реализует базовую функциональность, необходимую в элементе управления `Button`, что позволяет программистам выполнять наследование из этого класса и создавать собственные нестандартные элементы управления `Button`.

Пространство имен `System.Windows.Forms` предоставляет три элемента управления, производные от класса `ButtonBase`: `Button`, `CheckBox` и `RadioButton`. Этот раздел посвящен элементу управления `Button` (стандартной, хорошо известной прямоугольной кнопке). Остальные два элемента управления рассматриваются в последующих разделах этой главы.

Элемент управления Button присутствует практически в каждом диалоговом окне Windows, которое можно себе представить. В основном кнопка служит для решения трех видов задач.

- Закрытие диалогового окна с определенным состоянием (например, кнопки OK и Cancel (Отмена)).
- Выполнение действия с данными, введенными в диалоговом окне (например, при щелчке на кнопке Search (Поиск) после ввода критерия поиска).
- Открытие другого диалогового окна или приложения (например, кнопки Help (Справка)).

Работа с элементом управления Button очень проста. Обычно она сводится к добавлению элемента управления в форму и двойному щелчку на нем для добавления кода в событие Click — как правило, этого достаточно для большинства приложений.

Свойства элемента управления Button

В этом разделе рассматриваются некоторые свойства элемента управления Button. В результате вы получите представление о том, как его использовать. Наиболее часто применяемые свойства класса Button (даже если в действительности они определены в базовом классе ButtonBase) перечислены в табл. 15.3. Для ознакомления с полным перечнем свойств обращайтесь к документации .NET Framework SDK.

Таблица 15.3. Часто используемые свойства класса Button

Свойство	Описание
FlatStyle	Изменяет стиль кнопки. Если в качестве стиля установлен Popup, кнопка выглядит плоской до тех пор, пока пользователь не поместит указатель мыши над ней. В этом случае внешний вид кнопки изменяется на трехмерный
Enabled	Хотя это свойство наследуется из класса Control, оно указано в этой таблице, поскольку очень важно для кнопки. Установка его в false означает, что кнопка становится затемненной и щелчок на ней не вызывает никаких последствий
Image	Указывает изображение (растровое, значок и т.п.), которое будет отображаться на кнопке
ImageAlign	Указывает позицию изображения на кнопке

События элемента управления Button

До сих пор наиболее часто используемым событием кнопки было Click. Это событие происходит при каждом щелчке пользователя на кнопке — т.е. при нажатии и отпускании левой кнопки мыши в то время, когда указатель располагается над кнопкой. Следовательно, если нажать левую кнопку мыши и отвести указатель мыши в сторону от кнопки до освобождения кнопки мыши, событие Click не запускается. Кроме того, событие Click происходит, когда кнопка находится в фокусе и пользователь нажимает клавишу <Enter>. Это событие всегда нужно обрабатывать при наличии кнопки в форме.

В следующем практическом занятии мы создадим диалоговое окно с тремя кнопками. Две из них изменяют язык интерфейса с английского на датский и обратно. (Можете использовать любые языки по своему выбору.) Последняя кнопка закрывает диалоговое окно.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с кнопками

Чтобы создать небольшое Windows-приложение, которое использует три кнопки для изменения текста в заголовке диалогового окна, выполните следующие действия.

1. Создайте новое Windows-приложение `ButtonTest` в каталоге `C:\BegVCSharp\Chapter15`.
2. Разверните панель инструментов Toolbox, щелкнув на значке с канцелярской кнопкой, расположенной рядом с символом x в верхнем правом углу окна, и три раза выполните двойной щелчок на элементе управления `Button`. Разместите кнопки и измените размеры формы, как показано на рис. 15.3.
3. Щелкните на кнопке правой кнопкой мыши и в контекстном меню выберите пункт Properties (Свойства). Измените имя каждой кнопки, как показано на рис. 15.3, выбирая в окне Properties поле редактирования `(Name)` и вводя соответствующий текст.
4. Измените свойство `Text` каждой кнопки в соответствии с ее именем, но опустите при этом префикс `button`.
5. Чтобы было понятно, к какому языку выполняется переход, перед текстом желательно отображать соответствующий флаг. Выберите кнопку English (Английский) и найдите свойство `Image` (Изображение). Щелкните справа от него, чтобы открыть диалоговое окно, в котором можно будет добавить изображения в файл ресурсов формы. Щелкните на кнопке Import (Импортировать) и выполните просмотр доступных значков. Нужные нам значки включены в состав файлов проекта `ButtonTest`. Выберите значки `UK.PNG` и `DK.PNG`.
6. Выберите `UK` и щелкните на кнопке OK. Затем выберите кнопку `buttonDanish`, щелкните на поле `(...)` свойства `Image` и выберите `DK`, прежде чем щелкнуть на кнопке OK.
7. На этом этапе текст и значок кнопки размещены друг поверх друга, поэтому нужно изменить способ выравнивания значка. Для обеих кнопок English и Danish измените значение свойства `ImageAlign` на `MiddleLeft`.
8. Возможно, ширину кнопок придется изменить, чтобы текст начинался не сразу за изображениями. Для этого выберите каждую из кнопок и раздвиньте селекторную метку, расположенную у правого края кнопки.
9. И, наконец, щелкните на форме и измените свойство `Text` на `Do you speak English?` (Вы говорите по-английски?)

Вот и все, что касается интерфейса пользователя этого диалогового окна. Теперь диалоговое окно должно выглядеть подобно показанному на рис. 15.4.

Рис. 15.3. *Начальная компоновка пользовательского интерфейса*

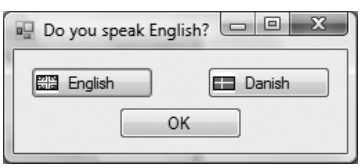


Рис. 15.4. *Финальная компоновка пользовательского интерфейса*

Теперь можно приступать к добавлению обработчиков событий. Дважды щелкните на кнопке English. Это приведет непосредственно к обработчику события элемента управления, используемому по умолчанию — для кнопки таким событием является событие `Click`, поэтому именно его обработчик создается.

Добавление обработчиков событий

Дважды щелкните на кнопке English и добавьте следующий код в обработчик события:

```
private void buttonEnglish_Click(object sender, EventArgs e)
{
  Text = "Do you speak English?";
}
```

Когда Visual Studio создает метод для обработки такого события, в качестве имени метода назначается имя элемента управления, за которым следуют символ подчеркивания и имя обрабатываемого события.

Первый параметр события `Click — object sender` — содержит элемент управления, на котором был выполнен щелчок. В данном примере им всегда будет элемент управления, указанный именем метода. В других случаях для обработки события многие элементы управления могут использовать один и тот же метод. В таких случаях для выяснения того, какой элемент управления вызывается, можно проверить это значение. Применение одного и того же метода для нескольких элементов управления описано в разделе "Элемент управления `TextBox`" далее в главе. Второй параметр — `System.EventArgs e` — хранит информацию о том, что произошло в действительности. В данном случае эта информация не требуется.

Вернитесь к представлению визуального конструктора и дважды щелкните на кнопке Danish. Откроется обработчик события этой кнопки. Его код должен иметь следующий вид:

```
private void buttonDanish_Click(object sender, EventArgs e)
{
  Text = "Taler du dansk?";
}
```

Этот метод идентичен методу `btnEnglish_Click`, за исключением того, что текст отображается на датском языке. И в заключение добавьте обработчик кнопки OK, как это уже делалось дважды. Однако теперь код несколько отличается от использованного ранее:

```
private void buttonOK_Click(object sender, EventArgs e)
{
  Application.Exit();
}
```

На этом создание кода приложения, а вместе с ним и первого примера, завершено. Скомпилируйте и запустите его, и пощелкайте на кнопках. Вы увидите, как будет изменяться текст в строке заголовка окна.

Элементы управления `Label` и `LinkLabel`

Вероятно, элемент управления `Label` является наиболее часто используемым. Он присутствует практически в каждом диалоговом окне любого Windows-приложения. Этот элемент управления служит только одной цели — отображению текста внутри формы.

В .NET Framework включены два по-разному отображаемых элемента управления типа метки:

- `Label` — стандартная метка Windows;
- `LinkLabel` — метка, аналогичная стандартной (и производная от нее), но отображаемая в виде гиперссылки.

Обычно для стандартной метки Label добавление кода обработки события не требуется, хотя она и поддерживает события, как и все элементы управления. Однако в случае LinkLabel требуется дополнительный код, чтобы посредством щелчка на этом элементе управления пользователи могли переходить по целевой ссылке.

Для элемента управления Label можно определять множество свойств. Большинство из них унаследованы от элемента управления Control, но некоторые являются новыми. Наиболее часто используемые свойства перечислены в табл. 15.4. Если не указано иное, свойства являются общими для элементов управления Label и LinkLabel.

Таблица 15.4. Часто используемые свойства элементов управления Label и LinkLabel

Свойство	Описание
BorderStyle	Указывает стиль рамки вокруг метки. По умолчанию рамка отсутствует
FlatStyle	Определяет способ отображения элемента управления. Установка значения этого свойства равным Popup приводит к тому, что элемент выглядит плоским до тех пор, пока пользователь не помещает указатель мыши над ним. При этом элемент приобретает приподнятый вид
Image	Указывает одиночное изображение (растровое, значок и т.п.), которое будет отображаться в элементе метки
ImageAlign	Указывает позицию отображения изображения в элементе Label
LinkArea	(Только для элемента управления LinkLabel.) Это свойство указывает часть текста, которая должна отображаться в качестве ссылки
LinkColor	(Только для элемента управления LinkLabel.) Это свойство задает цвет ссылки
Links	(Только для элемента управления LinkLabel.) Элемент управления LinkLabel может содержать более одной ссылки. Это свойство позволяет выбрать нужную ссылку. Элемент управления отслеживает ссылки, отображенные в тексте. Свойство недоступно во время разработки
LinkVisited	(Только для элемента управления LinkLabel.) Установка этого свойства равным true означает, что цвет ссылки изменяется после щелчка на ней
TextAlign	Указывает позицию отображения текста в элементе управления
VisitedLinkColor	(Только для элемента управления LinkLabel.) Это свойство указывает цвет элемента управления LinkLabel после щелчка на нем

Элемент управления TextBox

Текстовые поля нужно использовать в тех случаях, когда пользователям требуется предоставить возможность вводить текст, о котором ничего не известно во время разработки (например, имя пользователя). Основное назначение текстового поля — обеспечение возможности ввода текста. Хотя допускается ввод любых символов, пользователей можно ограничить вводом только числовых значений.

В .NET Framework предлагается два основных элемента управления для приема текста от пользователей: TextBox и RichTextBox. Оба эти элемента управления являются производными от базового класса TextBoxBase, который сам является производным от класса Control.

TextBoxBase предоставляет основные функциональные возможности манипулирования текстом в текстовом поле, такие как выбор текста, вырезание и вставка из буфера обмена, и широкий спектр событий. Пока не будем особо вникать в то, какие свойства наследуются из того или иного класса, а рассмотрим более простой из этих двух элементов управления — TextBox. Вначале создадим пример, демонстрирующий свойства элемента TextBox, а затем расширим его для демонстрации свойств RichTextBox.

Свойства элемента управления TextBox

Существует слишком много свойств, чтобы описывать их все. В табл. 15.5 перечислены наиболее часто используемые из них.

Таблица 15.5. Часто используемые свойства элемента управления TextBox

Свойство	Описание
CausesValidation	Когда элемент управления, у которого это свойство установлено в true, готов принять фокус, запускаются два события: Validating и Validated. Обработку этих свойств можно выполнять для проверки достоверности данных в элементе управления, который утрачивает фокус. В результате может возникать ситуация, когда элемент управления никогда не получит фокус. Соответствующие события освещены ниже
CharacterCasing	Значение, указывающее, изменяет ли элемент управления TextBox регистр введенного текста. Возможные значения следующие: Lower — весь введенный текст преобразуется в нижнему регистру Normal — текст не подвергается никаким изменениям Upper — весь введенный текст преобразуется к верхнему регистру
MaxLength	Значение, которое указывает максимальную длину (в символах) любого текста, введенного в элементе управления TextBox. Если максимальная длина должна ограничиваться только доступным объемом памяти, установите это значение равным нулю
Multiline	Указывает, является ли данный элемент многострочным, т.е. может ли отображать несколько строк текста. Когда это свойство установлено в true, обычно значение свойства WordWrap также устанавливают равным true
PasswordChar	Указывает, должен ли символ пароля замещать реальные символы, введенные в однострочном элементе TextBox. Если значение свойства Multiline установлено в true, это свойство не оказывает никакого влияния
ReadOnly	Булевское значение, указывающее, является ли текст доступным только для чтения
ScrollBars	Указывает, должен ли многострочный элемент управления TextBox отображать линейки прокрутки
SelectedText	Текст, который выбран в элементе управления TextBox
SelectionLength	Количество символов, выбранных в тексте. Если это значение больше общего числа символов в тексте, элемент управления переустанавливает его равным общему количеству символов минус значение свойства SelectionStart
SelectionStart	Начало выбранного текста в элементе управления TextBox
WordWrap	Указывает, должен ли многострочный элемент TextBox автоматически переносить слова на следующую строку, если длина строки превышает ширину элемента управления

События элемента управления TextBox

Тщательность проверки достоверности текста, введенного в элементах управления TextBox формы может оказывать решающее влияние на то, будут пользователи удовлетворены или же раздосадованы результатом. Возможно, вы на собственном опыте убеждались, насколько неприятно, когда достоверность содержимого диалогового окна проверяется только после щелчка на кнопке OK. Обычно такой подход к проверке достоверности данных ведет к отображению окна сообщения о том, что данные в "текстовом поле номер 3" неверны. В этом случает можно продолжать щелкать на кнопке OK до тех пор, пока все данные не будут введены правильно. Очевидно, что такой подход к проверке достоверности данных — не самый рациональный. Но что можно предпринять?

Ответ заключается в обработке событий проверки достоверности данных, предоставляемых элементом управления TextBox. Чтобы запретить ввод в текстовом поле недопустимых символов или гарантировать ввод значений только из допустимого диапазона, необходимо указать пользователю, достоверно ли введенное значение.

Список событий, предоставляемых элементом управления TextBox, приведен в табл. 15.6 (все они унаследованы от элемента управления Control).

Таблица 15.6. Часто используемые события элемента управления TextBox

Событие	Описание
Enter Leave Validating Validated	Эти четыре события происходят в указанном в этой таблице порядке. Они называются *событиями фокуса* и, за двумя исключениями, запускаются во всех случаях, когда фокус элемента управления изменяется. События Validating и Validated запускаются, только если свойство CausesValidation элемента управления, который принимает фокус, установлено в true. Принимающий фокус элемент запускает событие потому, что в некоторых случаях элемент управления не нужно проверять даже при изменениях фокуса. Пример такой ситуации — щелчок на кнопке Help (Справка).
KeyDown KeyPress KeyUp	Эти три события называют *событиями клавиш*. Они позволяют отслеживать и изменять текст, введенный в элементах управления. События KeyDown и KeyUp принимают код нажатой клавиши. Это позволяет выявлять нажатие специальных клавиш, таких как <Shift>, <Ctrl> или <F1>. Событие KeyPress принимает символ, соответствующий клавише клавиатуры. Это означает, что значение буквы *а* не совпадает со значением буквы *А*. Это удобно, если нужно исключить диапазон символов — например, разрешить ввод только числовых значений.
TextChanged	Происходит при каждом изменении текста в текстовом поле, независимо от сути изменения.

В следующем практическом занятии создается диалоговое окно, в котором можно вводить имя, адрес, род занятий и возраст. Назначение этого примера в том, чтобы получить необходимые навыки в манипулировании свойствами и использовании событий, а не построить какого-то действительно полезное приложение.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с элементом управления TextBox

Вначале создадим интерфейс пользователя.

1. Создайте новое Windows-приложение TextBoxControls в каталоге C:\BegVCSharp\Chapter15.

2. Перетащите элементы управления `Label`, `TextBox` и `Button` в область конструирования, чтобы получить форму, показанную на 15.5. Прежде чем размеры двух элементов управления `TextBox` — `textBoxAddress` и `textBoxOutput` — можно будет изменить, как показано на рисунке, значение их свойства `Multiline` потребуется установить в `true`. Для этого щелкните на элементах управления правой кнопкой мыши и в контекстном меню выберите пункт Properties.
3. Назовите элементы управления, как показано на рис. 15.5.

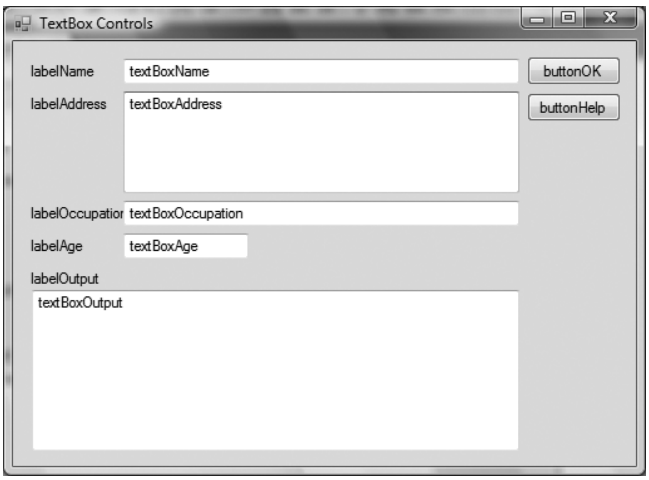


Рис. 15.5. *Пользовательский интерфейс приложения* `TextBoxControls`

4. Установите свойство `Text` всех остальных элементов управления в соответствии с их именами, за исключением префиксов, указывающих тип элемента (т.е. `Button`, `TextBox` и `Label`). Свойство `Text` формы установите в `TextBox Controls`.
5. Установите свойство `Scrollbars` элементов управления `textBoxOutput` и `textBoxAddress` в `Vertical`.
6. Установите свойство `ReadOnly` элемента управления `textBoxOutput` в `true`.
7. Установите свойство `CausesValidation` кнопки `btnHelp` в `false`. Вспомните из описания событий `Validating` и `Validated`, что установка этого свойства равным `false` позволяет пользователям щелкать на этой кнопке, не беспокоясь о достоверности введенных данных.
8. После того, как размеры формы и элементов управления подобраны, следует определить привязку элементов управления так, чтобы они вели себя должным образом при изменении размеров формы. Установите значение свойства `Anchor` для каждого типа элемента управления. Вначале, удерживая нажатой клавишу <Ctrl>, последовательно выберите все элементы управления типа `TextBox`, за исключением `textBoxOutput`. Выбрав все эти элементы управления, в окне Properties установите значение свойства `Anchor` в `Top,Left,Right`. Тем самым вы установите значение свойства `Anchor` для всех выбранных элементов типа `TextBox`. Выберите элемент управления `textBoxOutput` и установите значение его свойства `Anchor` в `Top,Bottom,Left,Right`. Теперь установите значение свойства `Anchor` обоих элементов управления `Button` в `Top,Right`.

Причина того, что элемент управления `textBoxOutput` привязывается, а не пристыковывается к нижнему краю формы, состоит в том, что размер области вывода текста должен изменяться при растягивании формы. Стыковка элемента управления с нижним краем формы приводила бы к его перемещению синхронно с нижним краем формы, но не к изменению его размеров.

9. Теперь остается определить последние свойства. Найдите свойства `Size` и `MinimumSize` формы. Форма будет практически бесполезной, если ее размеры меньше используемых в настоящий момент. Поэтому установите значение свойства `MinimumSize` равным значению `Size`.

Описание работы

Работа по определению визуальной части формы завершена. При выполнении приложения щелчок на кнопках или ввод текста не приведут ни к какому видимому результату, но при развертывании или растягивании диалогового окна элементы управления будут вести себя надлежащим образом, сохраняя свое взаимное расположение и изменяя размеры для заполнения всей области диалогового окна.

Добавление обработчиков событий

В представлении визуального конструктора дважды щелкните на кнопке `buttonOK`. Повторите это действие для второй кнопки. Как было показано в примере создания кнопки, ранее в этой главе, это ведет к созданию обработчиков события `Click`. Щелчок на кнопке OK должен приводить к передаче текста из текстовых полей ввода в поле вывода, доступное только для чтения.

Код двух обработчиков событий `Click` имеет следующий вид:

```
private void buttonOK_Click(object sender, EventArgs e)
{
   // Проверка достоверности значений не выполняется,
   // поскольку это не обязательно.

   string output;

   // Конкатенация текстовых значений из четырех элементов управления типа TextBox.
   output = "Name: " + textBoxName.Text + "\r\n";
   output += "Address: " + textBoxAddress.Text + "\r\n";
   output += "Occupation: " + textBoxOccupation.Text + "\r\n";
   output += "Age: " + textBoxAge.Text;

   // Вставка нового текста.
   textBoxOutput.Text = output;
}

private void buttonHelp_Click(object sender, EventArgs e)
{
   // Запись краткого описания каждого элемента TextBox в поле textBoxOutput.
   string output;

   output = "Name = Your name\r\n";
   output += "Address = Your address\r\n";
   output += "Occupation = Only allowed value is 'Programmer'\r\n";
   output += "Age = Your age";

   // Вставка нового текста.
   this.textBoxOutput.Text = output;
}
```

Фрагмент кода ***Chapter15\TextBoxControls\Form1.cs***

Обе функции используют свойство Text каждого элемента управления TextBox. Свойство Text элемента управления textBoxAge служит для получения значения, введенного в качестве возраста данного лица, а это же свойство элемента управления textBoxOutput — для отображения объединенного текста.

Вставка информации, введенной пользователем, выполняется без проверки ее правильности. Следовательно, проверка должна осуществляться где-то в другом месте программы. В данном примере правильность введенных значений определяют следующие критерии.

- Имя пользователя не может быть пустым.
- Возраст пользователя должен быть числом, которое больше или равно нулю.
- Профессией пользователя должна быть Programmer (программист) или же это значение должно быть оставлено пустым.
- Адрес пользователя не может быть пустым.

Следовательно, проверка, которая должна выполняться для двух текстовых полей (textBoxName и textBoxAddress), идентична. Кроме того, необходимо воспрепятствовать вводу недопустимого значения в поле Age (Возраст), а также проверить, подтверждает ли пользователь, что он является программистом.

Чтобы запретить щелчок на кнопке OK до ввода каких-либо данных, начнем с установки ее свойства Enabled в false — на этот раз в визуальном конструкторе формы, а не в окне Properties. При установке свойств в визуальном конструкторе это должно выполняться не ранее вызова сгенерированного кода в функции InitializeComponent():

```
public Forml()
{
  InitializeComponent();
  buttonOK.Enabled = false;
}
```

Теперь создадим обработчик для двух текстовых полей, которые необходимо проверять на отсутствие пустых значений. Это выполняется посредством подписки на событие Validating текстовых полей. Поскольку одна и та же операция должна проводиться с обоими элементами управления, им назначается один и тот же обработчик события. Выберите в форме оба элемента управления и в списке событий выберите событие Validating, введя textBoxEmpty_Validating в качестве имени события. Для открытия списка событий в окне Properties достаточно щелкнуть на кнопке со значком молнии.

В отличие от ранее рассмотренного обработчика события кнопки, обработчик события Validating является специализированной версией стандартного обработчика System.EventHandler. Это событие нуждается в специальном обработчике, поскольку в случае отрицательного результата проверки требуется способ предотвращения любой дальнейшей обработки. Отмена дальнейшей обработки, по сути, означала бы невозможность выхода из текстового поля до момента ввода допустимых данных.

Использование событий Validating и Validated в сочетании со свойством CausesValidation позволяет решить неприятную проблему, возникавшую при использовании событий GotFocus и LostFocus для проверки достоверности элементов управления в предшествующих версиях Visual Studio. Проблема возникала при непрерывном генерировании событий GotFocus и LostFocus вследствие того, что проверяемый код пытался перемещать фокус между элементами управления, что создавало бесконечный цикл.

Замените оператор throw в обработчике события, сгенерированном программой Visual Studio, следующим кодом:

```
private void textBoxEmpty_Validating (object sender,
                              System.ComponentModel.CancelEventArgs e)
{
  TextBox tb = (TextBox)sender;
```

```
    if (tb.Text.Length == 0)
      tb.BackColor = Color.Red;
    else
      tb.BackColor = System.Drawing.SystemColors.Window;
    ValidateOK();
}
```

Поскольку этот метод используется для обработки события более чем одним текстовым полем, неизвестно, какое из них вызывает функцию. Однако известно, что результат вызова метода должен быть одинаковым, независимо от вызывающего объекта. Поэтому можно просто привести параметр `sender` к типу `TextBox`:

```
TextBox tb = (TextBox)sender;
```

Если длина текста в текстовом поле равна нулю, цвет фона нужно установить красным, а значение свойства `Tag` — равным `false`. В противном случае в качестве цвета фона устанавливается стандартный цвет окна Windows.

Для установки стандартного цвета в элементе управления всегда следует использовать цвета, определенные в перечислении `System.Drawing.SystemColors`. Если просто установить цвет белым, приложение будет выглядеть странно в случае изменения пользователем заданных по умолчанию цветовых настроек.

Описание функции `ValidateOK()` приведено в конце этого примера. Следующий обработчик события `Validating`, который необходимо добавить, предназначен для текстового поля Occupation (Профессия). Процедура добавления полностью аналогична описанной для двух предыдущих обработчиков, но код проверки достоверности отличается, поскольку, чтобы быть допустимым, значение профессии должно быть `Programmer` (Программист) либо пустой строкой. Чтобы добавить обработчик события, дважды щелкните на событии `Validating` элемента управления `textBoxOccupation`.

Теперь можно добавить сам обработчик:

```
private void textBoxOccupation_Validating(object sender,
                              System.ComponentModel.CancelEventArgs e)
{
  TextBox tb = (TextBox)sender;

  if (tb.Text == "Programmer" || tb.Text.Length == 0)
    tb.BackColor = System.Drawing.SystemColors.Window;
  else
    tb.BackColor = Color.Red;
  ValidateOK();
}
```

Предпоследняя задача, которую остается решить — предоставление обработчика для текстового поля Age (Возраст). Необходимо, чтобы пользователи вводили только положительные числа (включая 0, для упрощения проверки). Для этого мы используем событие `KeyPress`, чтобы удалить любые нежелательные символы до того, как они будут отображены в текстовом поле. Кроме того, мы ограничим количество символов, которые могут быть введены в элементе управления, тремя.

Вначале установите значение `MaxLength` элемента управления `textBoxAge` равным 3. Затем подпишитесь на событие `KeyPress` посредством двойного щелчка на событии `KeyPress` в списке Events окна Properties. Обработчик события `KeyPress` также является специализированным.

Мы предоставляем обработчик `System.Windows.Forms.KeyPressEventHandler`, поскольку событие нуждается в информации о нажатой клавише.

В сам обработчик события понадобится добавить следующий код:

```
private void textBoxAge_KeyPress(object sender, KeyPressEventArgs e)
{
   if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8)
      e.Handled = true;
}
```

ASCII-значения для символов от 0 до 9 лежат в диапазоне от 48 до 57, поэтому нужно удостовериться, что введенный символ относится к этому диапазону — за одним исключением. ASCII-значение 8 представляет клавишу забоя, и в целях упрощения редактирования это поведение следует оставить без изменений. Установка значения свойства Handled объекта KeyPressEventArgs в true указывает элементу управления, что он не должен выполнять с символом никаких других действий. Поэтому если нажатая клавиша не является клавишей цифры или забоя, она не отображается.

На данный момент элемент управления не помечен как недопустимый или допустимый. Это связано с тем, что для определения того, было ли что-то введено в элементе управления, требуется выполнение еще одной проверки. Это не представляет сложности, поскольку метод для такой проверки уже создан. В списке событий выберите обработчик события textBoxEmpty_Validating из раскрывающегося списка события Validating элемента управления textBoxAge.

Остается определить последний компонент — метод ValidateOK, который активизирует или отключает кнопку OK:

```
private void ValidateOK()
{
   buttonOK.Enabled = (textBoxName.BackColor != Color.Red &&
   textBoxAddress.BackColor != Color.Red &&
   textBoxOccupation.BackColor != Color.Red &&
   textBoxAge.BackColor != Color.Red);
}
```

Этот метод просто устанавливает значение свойства Enabled кнопки OK в true, если значения всех свойств Tag равны true.

Если теперь протестировать программу, результат должен выглядеть подобно показанному на рис. 15.6 (естественно, без красного фона). Обратите внимание, что можно щелкнуть на кнопке Help (Справка), находясь в текстовом поле с недопустимыми данными, и при этом цвет фона не будет изменяться на красный.

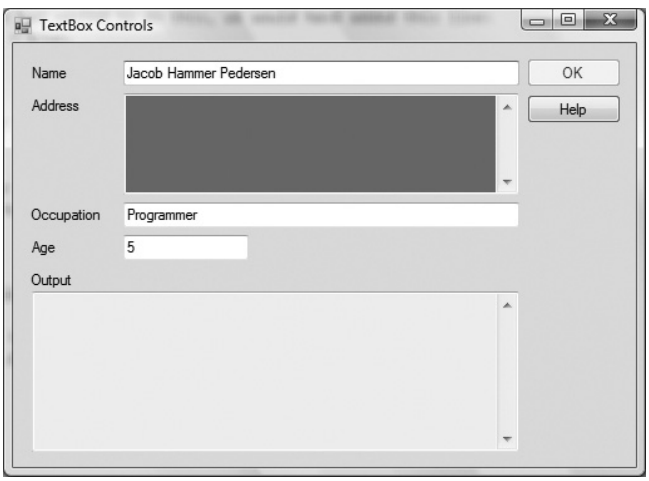


Рис. 15.6. *Приложение* `TextBoxControls` *в действии*

Элементы управления RadioButton и CheckBox

Как уже упоминалось, элементы управления RadioButton и CheckBox имеют тот же самый базовый класс, что и элемент управления Button, хотя по внешний виду и использованию они существенно отличаются.

Традиционно элементы управления RadioButton (переключатели) отображаются в виде метки с маленькой окружностью слева от нее, которая может быть выбрана или не выбрана. Переключатели следует применять, когда пользователю нужно предоставить возможность выбора между двумя или более взаимоисключающими опциями — например пол.

Чтобы сгруппировать переключатели для образования единого логического блока, необходимо использовать элемент управления GroupBox или другой контейнер. При первоначальном помещении элемента GroupBox на форму и последующем размещении необходимых элементов управления RadioButton внутри его границ, состояние элементов RadioButton будет автоматически изменено для отражения того, что только одна опция внутри групповой рамки может быть выбрана. Если элементы управления RadioButton размещены не внутри элемента управления GroupBox, на *форме* в любой момент времени может быть выбран только один из них.

Элемент управления CheckBox (флажок) отображается в виде метки с расположенным слева от него маленьким квадратом. Флажок должен применяться, когда пользователю нужно предоставить возможность выбора одной или более опций — например, для заполнения вопросника об использовавшихся ранее операционных системах (Windows XP, Windows Vista, Windows 7, Linux и т.п.).

После ознакомления с важными свойствами и событиями этих двух элементов управления, начиная с RadioButton, мы рассмотрим краткий пример их использования.

Свойства элемента управления RadioButton

Поскольку элемент управления RadioButton является производным от ButtonBase и т.к. общие свойства уже были рассмотрены в примере использования элемента управления Button, остается рассмотреть всего несколько свойств (они перечислены в табл. 15.7). Полный перечень свойств приведен в документации .NET Framework SDK.

Таблица 15.7. Часто используемые свойства элемента управления RadioButton

Свойство	Описание
Appearance	Переключатель может отображаться в виде метки с круглой меткой выбора слева, в середине или справа от нее, либо же в виде стандартной кнопки. При отображении в виде стандартной кнопки выбранный элемент управления выглядит нажатым, а не выбранный — отжатым
AutoCheck	Когда значение этого свойства равно true, при щелчке на переключателе в окружности выбора отображается черная точка. Когда это значение равно false, установка флажка переключателя должна выполняться в коде вручную из обработчика события Click
CheckAlign	Это свойство используется для изменения способа выравнивания части переключателя, определяющей флажок. Его значение по умолчанию — ContentAlignment.MiddleLeft
Checked	Указывает состояние элемента управления. Значение этого свойства равно true, если черная точка отображается в элементе управления. В противном случае оно равно false

События элемента управления RadioButton

Как правило, при работе с элементами управления RadioButton придется использовать только одно событие, но возможна подписка и на множество других. В этой главе рассматриваются только два события, описанные в табл. 15.8, причем второе упомянуто лишь для того, чтобы обратить внимание на незначительное различие между ними.

Таблица 15.8. Часто используемые события элемента управления RadioButton

Событие	Описание
CheckedChanged	Отправляется при изменении состояния выбора элемента управления RadioButton
Click	Отправляется при каждом щелчке на элементе управления RadioButton. Это событие не эквивалентно событию CheckedChange, поскольку при двух и более щелчках на элементе управления RadioButton свойство Checked изменяется только один раз — и только, если оно еще не было выбрано. Более того, если значение свойства AutoCheck кнопки, на которой выполняется щелчок, равно false, кнопка вообще не будет выбрана и отправляться будет только событие Click

Свойства элемента управления CheckBox

Несложно догадаться, что свойства и события этого элемента управления подобны свойствам и событиям элемента управления RadioButton, но в табл. 15.9 перечислены два новых свойства.

Таблица 15.9. Часто используемые свойства элемента управления CheckBox

Свойство	Описание
CheckState	В отличие от переключателя, флажок может пребывать в трех состояниях: Checked, Indeterminate и Unchecked. Когда состояние флажка — Indeterminate, индикатор состояния флажка, расположенный рядом меткой, обычно затемнен, указывая, что либо текущее значение флажка недопустимо, либо оно не может быть определено по какой-либо причине (например, флажок указывает состояние “только для чтения” для двух выбранных файлов, из которых один доступен только для чтения, а второй — нет), либо не имеет смысла в данной ситуации
ThreeState	Когда значение этого свойства — false, пользователь не сможет изменять состояние CheckState на Indeterminate. Однако значение свойства CheckState все же можно изменять на Indeterminate в коде

События элемента управления CheckBox

Обычно в этом элементе управления придется использовать одно или два события. Хотя событие CheckChanged применяется и в RadioButton, и в CheckBox, его действие различно в этих двух элементах управления. События элемента управления CheckBox описаны в табл. 15.10.

На этом ознакомление с событиями и свойствами элементов управления RadioButton и CheckBox завершено. Но прежде чем приступить к рассмотрению примера их использования, рассмотрим упомянутый ранее элемент управления GroupBox.

Таблица 15.10. События элемента управления `CheckBox`

Событие	Описание
`CheckedChanged`	Происходит при каждом изменении свойства `Checked` флажка. Обратите внимание, что в элементе управления `CheckBox`, свойство `ThreeState` которого имеет значение `true`, можно выполнить щелчок на флажке без изменения значения свойства `Checked`. Это происходит при изменении состояния флажка с `Checked` на `Indeterminate`
`CheckStateChanged`	Происходит при каждом изменении свойства `CheckedState` флажка. Поскольку и `Checked`, и `Unchecked` — возможные значения свойства `CheckedState`, это событие отправляется при каждом изменении свойства `Checked`. Кроме того, оно отправляется также при изменении состояния с `Checked` на `Indeterminate`

Элемент управления `GroupBox`

Элемент управления `GroupBox` часто применяют для логического группирования набора элементов управления, таких как `RadioButton` и `CheckBox`, и для отображения заголовка и рамки вокруг этого набора.

Использование групповой рамки сводится к ее перетаскиванию на форму и последующему перетаскиванию на нее тех элементов управления, которые она должна содержать (но не в обратном порядке — т.е. нельзя наложить групповую рамку поверх уже существующих элементов управления). В результате родительским элементом для элементов управления становится групповая рамка, а не форма, что делает возможным наличие в форме более одного выбранного переключателя в любой конкретный момент времени. Однако внутри групповой рамки может быть выбран только один переключатель.

Пожалуй, отношение между родительским и дочерним элементом требует дополнительного пояснения. Когда элемент управления помещается на форму, форма становится его родительским элементом и, следовательно, элемент управления является дочерним элементом формы. После помещения на форму элемент управления `GroupBox` становится дочерним элементом формы. Поскольку групповая рамка сама может содержать элементы управления, она становится их родительским элементом. В результате перемещение элемента управления `GroupBox` приводит к перемещению всех находящихся в ней элементов управления.

Еще одно следствие помещения элементов управления в групповую рамку состоит в том, что на них можно оказывать влияние, изменяя соответствующее свойство групповой рамки. Например, если нужно отключить все элементы управления внутри элемента управления `GroupBox`, можно просто установить значение свойства `Enabled` элемента `GroupBox` в `false`.

Применение элемента управления `GroupBox` продемонстрировано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование элементов управления `RadioButton` и `CheckBox`

В этом упражнении мы изменим пример `TextBoxControls`, созданный ранее. В том примере единственной допустимой профессией была программист. Вместо того чтобы вынуждать пользователей полностью вводить этот текст, изменим это текстовое поле на флажок. Для демонстрации применения переключателя предложим пользователю предоставить дополнительную информацию: указать свой пол.

Модифицируйте пример использования текстовых полей, как описано ниже.

1. Удалите метку `labelOccupation` и текстовое поле `textBoxOccupation`.
2. Добавьте элементы управления `CheckBox`, `GroupBox` и два элемента управления `RadioButton` и назовите их, как показано на рис. 15.7. В отличие от других использованных до сих пор элементов управления, элемент управления `GroupBox` находится в разделе Containers (Контейнеры) панели Toolbox.
3. Значения свойств `Text` элементов управления `RadioButton` и `CheckBox` должны совпадать с именами элементов управления без первых трех букв, а значением этого свойства элемента управления `GroupBox` должно быть `Sex`.

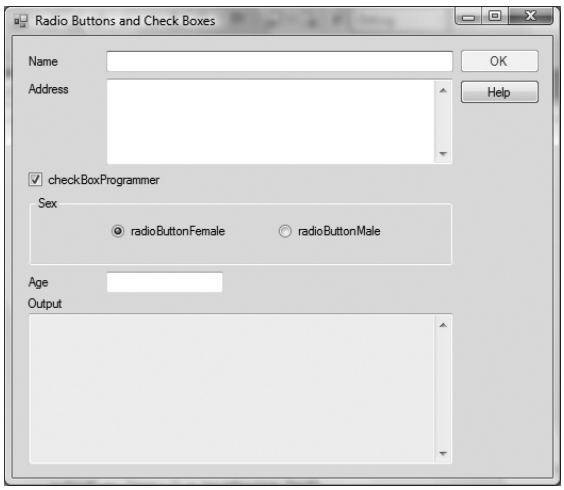


Рис. 15.7. *Пользовательский интерфейс приложения для работы с переключателями и флажками*

4. Установите значение свойства `Checked` флажка `checkBoxProgrammer` равным `true`. Обратите внимание, что значение свойства `CheckState` автоматически изменяется на `Checked`.
5. Установите значение свойства `Checked` переключателя `radioButtonMale` или `radioButtonFemale` в `true`. Обратите внимание, что установить в `true` значения этого свойства для обоих переключателей не получится. При попытке проделать это со второй кнопкой значение первого переключателя `RadioButton` автоматически изменится на `false`.

Реализация визуальных аспектов примера не требует никаких дополнительных действий, но в код понадобится внести ряд изменений. Сначала необходимо удалить все ссылки на удаленное текстовое поле. Модифицируйте существующий код следующим образом.

1. Из кода метода `ValidateOK` удалите проверку фона текстового поля `textBoxOccupation`.

```
private void ValidateOK()
{
   // Активизирует кнопку OK, если значения всех свойств Tag равны true.
   buttonOK.Enabled = (textBoxName.BackColor != Color.Red &&
                       textBoxAddress.BackColor != Color.Red &&
                       textBoxAge.BackColor != Color.Red);
}
```

Фрагмент кода ***Chapter15\Radio and Check Buttons\Form1.cs***

2. Полностью удалите метод textBoxOccupation_Validating.
3. Удалите ссылку из buttonOK_Click.

Описание работы

Поскольку теперь используется флажок, а не текстовое поле, известно, что пользователь не может ввести никакую недопустимую информацию, поскольку он всегда либо является программистом, либо не является им.

Известно также, что пользователь является либо мужчиной, либо женщиной, и поскольку значение свойства одного из переключателей установлено равным true, пользователь лишен возможности выбора недопустимого значения. Поэтому остается только изменить текст справки и вывод программы. Это выполняется в обработчиках событий кнопок:

```
private void buttonHelp_Click(object sender, System.EventArgs e)
{
   // Запись краткого описания каждого элемента TextBox в поле Output.
   string output;

   output = "Name = Your name\r\n";
   output += "Address = Your address\r\n";
   output += "Programmer = Check 'Programmer' if you are a programmer\r\n";
   output += "Sex = Choose your sex\r\n";
   output += "Age = Your age";

   // Вставка нового текста.
   this.textBoxOutput.Text = output;
}
```

Изменения претерпел только текст справки, поэтому метод help не таит никаких сюрпризов. Обработчик событий для кнопки OK представляет несколько больший интерес:

```
private void buttonOK_Click(object sender, EventArgs e)
{
   // Проверка достоверности значений не выполняется,
   // поскольку это не обязательно.

   string output;

   // Конкатенация текстовых значений четырех элементов управления типа TextBox.
   output = "Name: " + this.textBoxName.Text + "\r\n";
   output += "Address: " + this.textBoxAddress.Text + "\r\n";
   output += "Occupation: " + (string)( checkBoxProgrammer.Checked ?
             "Programmer": "Not a programmer") + "\r\n";
   output += "Sex: " + (string)( radioButtonFemale.Checked ? "Female":
             "Male") + "\r\n";
   output += "Age: " + this.textBoxAge.Text;

   // Вставка нового текста.
   this.textBoxOutput.Text = output;
}
```

Первая из выделенных строк — это строка, в которой выводится профессия пользователя. Мы исследуем свойство Checked элемента управления CheckBox, и если его значение равно true, выводится строка Programmer (Программист). Если это значение — false, выводится строка Not a programmer (Не программист).

Во второй выделенной строке кода анализируется только переключатель radioButtonFemale. Если значение свойства Checked этого элемента управления равно true, значит, пользователь — женщина. Если же оно — false, пользователь — мужчина. Во время запуска программы оба эти переключателя можно было бы оставлять не выбранными, но выбор одного из них во время разработки гарантирует выбор одного из этих переключателей при любых обстоятельствах.

Теперь при запуске примера результат должен быть подобным показанному на рис. 15.8.

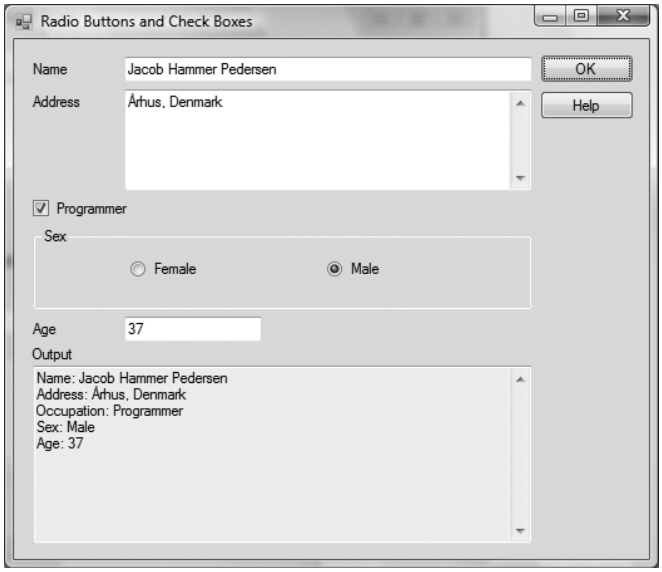


Рис. 15.8. *Приложение для работы с переключателями и флажками во время выполнения*

Элемент управления `RichTextBox`

Подобно обычному элементу управления `TextBox`, `RichTextBox` является производным от элемента управления `TextBoxBase`. Поэтому он имеет ряд свойств общих с `TextBox`, но различий значительно больше. В то время как `TextBox` обычно используется для получения коротких текстовых строк от пользователя, `RichTextBox` служит для отображения и ввода форматированного текста (например, **полужирного**, подчеркнутого и *курсивного*). Это достигается использованием стандарта форматированного текста, получившего название Rich Text Format (расширенный текстовый формат), или RTF.

В предыдущем примере применялся стандартный элемент управления `TextBox`. С таким же успехом можно было использовать `RichTextBox`. Фактически, как показано в последующем примере, вместо элемента управления `TextBox` с именем `textBoxOutput` можно вставить элемент `RichTextBox` с таким же именем, и пример будет вести себя точно так же, как ранее.

Свойства элемента управления `RichTextBox`

Поскольку этот вид текстового поля сложнее исследованного в предыдущем разделе, не удивительно, что он обладает большим количеством доступных для использования свойств. Наиболее часто используемые свойства элемента управления `RichTextBox` описаны в табл. 15.11.

Как видно из приведенного перечня, большинство новых свойств связано с выбором. Это обусловлено тем, что любое форматирование, которое будет выполнять пользователь в процессе работы с текстом, скорее всего, будет применяться к выделенному им фрагменту. Если фрагмент не выделен, форматирование начинается с позиции курсора в тексте, называемой *точкой вставки*.

Таблица 15.11. Часто используемые свойства элемента управления `RichTextBox`

Свойство	Описание
`CanRedo`	Значение этого свойства равно `true`, когда последняя отмененная операция может быть снова применена с помощью метода `Redo`
`CanUndo`	Значение этого свойства равно `true`, если возможна отмена последнего действия, выполненного по отношению к элементу управления `RichTextBox`. Обратите внимание, что свойство `CanUndo` определено в классе `TextBoxBase`, поэтому оно доступно также и для элементов управления `TextBox`
`RedoActionName`	Содержит имя действия, которое должно быть выполнено методом `Redo`
`DetectUrls`	Значение этого свойства необходимо установить в `true`, если требуется, чтобы элемент управления обнаруживал URL-адреса и форматировал их (подчеркивал, как это имеет место в браузере)
`Rtf`	Соответствует свойству `Text`, за исключением того, что содержит текст в RTF-формате
`SelectedRtf`	Служит для получения или установки текста, выбранного в элементе управления, в RTF-формате. При копировании этого текста в другое приложение — например, Word — он сохранит все форматирование
`SelectedText`	Как и `SelectedRtf`, это свойство можно использовать для получения или установки выбранного текста. Однако, в отличие от RTF-версии свойства, все форматирование утрачивается
`SelectionAlignment`	Представляет выравнивание выбранного текста. Свойство может принимать значения `Center`, `Left` или `Right`
`SelectionBullet`	Это свойство служит для определения того, должен ли выбранный текст содержать маркеры абзацев, а также для вставки и удаления маркеров
`BulletIndent`	Указывает количество пикселей отступа маркера
`SelectionColor`	Изменяет цвет текста в выделенном фрагменте
`SelectionFont`	Изменяет шрифт текста в выделенном фрагменте
`SelectionLength`	Устанавливает или извлекает длину выделенного фрагмента
`SelectionType`	Содержит информацию о выделенном фрагменте. Это свойство будет сообщать о том, выбран ли один или более объектов OLE, либо же только текст
`ShowSelectionMargin`	Если значение этого свойства равно `true`, слева от элемента управления `RichTextBox` будет отображаться граница. Она облегчает пользователю выбор текста
`UndoActionName`	Извлекает имя действия, которое будет использовано, если пользователь решит отменить что-либо
`SelectionProtected`	Установка значения этого свойства в `true` позволяет указать, что определенные фрагменты текста не должны изменяться

События элемента управления RichTextBox

Большинство событий, используемых элементом управления RichTextBox, совпадают с таковыми для элемента TextBox, но некоторые из новых свойств, представляющих интерес, перечислены в табл. 15.12.

Таблица 15.12. События элемента управления RichTextBox

Событие	Описание
LinkClicked	Отправляется, когда пользователь щелкает на ссылке внутри текста
Protected	Отправляется, когда пользователь пытается изменять текст, который помечен как защищенный
SelectionChanged	Отправляется при изменении выделенного фрагмента. Если по какой-либо причине изменение выделенного фрагмента пользователем нежелательно, это событие позволяет воспрепятствовать этому

В следующем практическом занятии мы создадим очень простой текстовый редактор. Пример демонстрирует способы изменения общего форматирования текста, а также способы загрузки и сохранения текста из элемента управления RichTextBox. Для простоты загрузка и сохранение выполняется в одном и том же фиксированном файле.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование элемента управления RichTextBox

Как всегда, начнем с разработки формы.

1. Создайте новое Windows-приложение Simple Text Editor в каталоге C:\BegVCSharp\Chapter15.
2. Создайте форму, как показано на рис. 15.9. Текстовое поле textBoxSize должно быть элементом управления типа TextBox. Текстовое поле RichTextBoxText должно быть элементом управления типа RichTextBox.

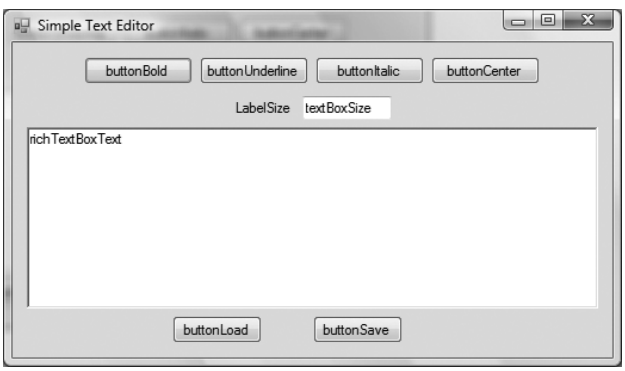


Рис. 15.9. Пользовательский интерфейс приложения Simple Text Editor

3. Назовите элементы управления, как показано на рис. 15.9.
4. Установите свойство Text всех элементов управления, кроме текстовых полей, в соответствии с их именами, за исключением первой части имени, описывающей тип элемента управления.

5. Измените значение свойства `Text` текстового поля `textBoxSize` на `10`.
6. Привяжите элементы управления, как показано в следующей таблице.

Имя элемента управления	Значение привязки
`buttonLoad` и `buttonSave`	`Bottom`
`richTextBoxText`	`Top, Left, Bottom, Right`
Все остальные	`Top`

7. Установите значение свойства `MinimumSize` формы таким же, как у свойства `Size`.

Описание работы

На этом реализация визуальной части примера завершена. Переходя непосредственно к работе с кодом, дважды щелкните на кнопке Bold (Полужирный), чтобы добавить в код обработчик события `Click`. Код обработчика события имеет следующий вид:

```
private void buttonBold_Click(object sender, EventArgs e)
{
  Font oldFont;
  Font newFont;

  oldFont = this.richTextBoxText.SelectionFont;

  if (oldFont.Bold)
    newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
  else
    newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

  this.richTextBoxText.SelectionFont = newFont;
  this.richTextBoxText.Focus();
}
```

Фрагмент кода ***Chapter15\Simple Text Editor\Form1.cs***

Начнем с получения шрифта, используемого в текущем выделенном фрагменте, и присваивания его локальной переменной `oldFont`. Затем проверим, не является ли этот выделенный фрагмент уже полужирным. Если да, атрибут полужирного шрифта следует удалить. В противном случае его нужно установить. Новый шрифт создается с применением `oldFont` в качестве прототипа, но с добавлением или удалением полужирного стиля при необходимости.

И, наконец, мы присваиваем новый шрифт выделенному фрагменту и возвращаем фокус элементу управления `RichTextBox`.

Обработчики событий для кнопок `buttonItalic` и `buttonUnderline` аналогичны рассмотренному выше, за исключением того, что они выполняют проверку на наличие соответствующих стилей. Дважды щелкните на кнопках Italic (Курсив) и Underline (Подчеркнутый) и добавьте следующий код:

```
private void buttonUnderline_Click(object sender, EventArgs e)
{
  Font oldFont;
  Font newFont;

  // Получение шрифта, используемого в выделенном тексте.
  oldFont = this.richTextBoxText.SelectionFont;

  // Если в настоящий момент шрифт использует подчеркнутый стиль, его нужно удалить.
  if (oldFont.Underline)
    newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
```

```
    else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

    // Вставка нового шрифта.
    this.richTextBoxText.SelectionFont = newFont;
    this.richTextBoxText.Focus();
}

private void buttonItalic_Click(object sender, EventArgs e)
{
    Font oldFont;
    Font newFont;

    // Получение шрифта, используемого в выделенном тексте.
    oldFont = this.richTextBoxText.SelectionFont;

    // Если в настоящий момент шрифт использует курсивный стиль, его нужно удалить.
    if (oldFont.Italic)
      newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
    else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

    // Вставка нового шрифта.
    this.richTextBoxText.SelectionFont = newFont;
    this.richTextBoxText.Focus();
}
```

Дважды щелкните на последней кнопке форматирования Center (По центру) и добавьте следующий код:

```
private void buttonCenter_Click(object sender, EventArgs e)
{
    if (this.richTextBoxText.SelectionAlignment == HorizontalAlignment.Center)
      this.richTextBoxText.SelectionAlignment = HorizontalAlignment.Left;
    else
      this.richTextBoxText.SelectionAlignment = HorizontalAlignment.Center;
    this.richTextBoxText.Focus();
}
```

В этом методе нужно обратиться к еще одному свойству — `SelectionAlignment`, чтобы проверить, центрирован ли уже текст в выделенном фрагменте. Это требуется для того, чтобы обеспечить поведение кнопки, подобное переключателю — если текст выровнен по центру, он становится выровненным по левому краю; в противном случае он выравнивается по центру. Свойство `HorizontalAlignment` представляет собой перечисление, которое может принимать значения `Left` (по левому краю), `Right` (по правому краю), `Center` (по центру), `Justify` (по ширине) и `NotSet` (без выравнивания). В данном случае мы просто проверяем, установлено ли значение `Center`. Если да, задается выравнивание по левому краю. В противном случае устанавливается выравнивание по центру.

Последняя задача, которую может выполнять наш простой текстовый редактор — установка размера текста. Мы добавим два обработчика событий для текстового поля `Size`: один для управления вводом и один для обнаружения момента завершения ввода значения пользователем.

В списке Events окна Properties найдите события `KeyPress` и `Validated` элемента управления `textBoxSize` и дважды щелкните на них, чтобы добавить в код обработчики этих событий.

В отличие от события `Validating`, использованного ранее, событие `Validated` происходит после завершения всей проверки. Оба события используют вспомогательный метод `ApplyTextSize`, который принимает строку, содержащую размер текста:

```
private void textBoxSize_KeyPress(object sender, KeyPressEventArgs e)
{
  if ((e.KeyChar < 48 || e.KeyChar > 57) &&
                                  e.KeyChar != 8 && e.KeyChar != 13)
    e.Handled = true;
  else if (e.KeyChar == 13)
  {
    TextBox txt = (TextBox)sender;
    if (txt.Text.Length > 0)
      ApplyTextSize(txt.Text);
    e.Handled = true;
    this.richTextBoxText.Focus();
  }
}
private void textBoxSize_Validated(object sender, EventArgs e)
{
  ApplyTextSize(((TextBox)sender).Text);
  this.richTextBoxText.Focus();
}
private void ApplyTextSize(string textSize)
{
  float newSize = Convert.ToSingle(textSize);
  FontFamily currentFontFamily;
  Font newFont;
  currentFontFamily = this.richTextBoxText.SelectionFont.FontFamily;
  newFont = new Font(currentFontFamily, newSize);
  this.richTextBoxText.SelectionFont = newFont; }
```

Событие `KeyPress` препятствует вводу пользователем любых значений кроме целочисленных и вызывает метод `ApplyTextSize`, если пользователь нажимает клавишу <Enter>. Интересующие нас действия выполняются во вспомогательном методе `ApplyTextSize()`. Он начинает свою работу с преобразования размера строки из `string` в `float`. Как уже было отмечено, приложение препятствует вводу пользователем чего-либо кроме целочисленных значений, но при создании нового шрифта требуется тип `float`, поэтому мы выполняем преобразование к нужному типу.

После этого мы извлекаем семейство, к которому принадлежит данный шрифт, и создаем новый шрифт этого же семейства, но с новым размером. И, наконец, мы устанавливаем новый шрифт в качестве шрифта выбранного текста.

Вот и все форматирование, которое можно выполнить, но кое-какие задачи решаются самим элементом управления `RichTextBox`. Если теперь запустить пример, можно будет определять для текста полужирный, курсивный и подчеркнутый шрифт и центрировать текст. Это именно то, что требовалось, но обратите внимание на еще одно интересное обстоятельство: попытайтесь ввести в тексте веб-адрес — например, `www.wrox.com`. Этот текст распознается элементом управления как адрес в Интернете и подчеркивается, а указатель мыши при нахождении над таким текстом изменяет свою форму на изображение руки. Если вы полагаете, что на этом тексте можно щелкнуть, чтобы перейти к соответствующей странице, то почти правы. Однако вначале понадобится предусмотреть обработку события, которое отправляется, когда пользователь щелкает на ссылке: `LinkClicked`.

Найдите событие `LinkClicked` в списке Events окна Properties и дважды щелкните на нем, чтобы добавить обработчик этого события в код. С этим обработчиком события вы еще не встречались — он используется для предоставления текста ссылки, на которой был выполнен щелчок. Данный обработчик удивительно прост:

```
private void richTextBoxText_LinkClicked (object sender,
                            System.Windows.Forms.LinkClickedEventArgs e)
{
  System.Diagnostics.Process.Start(e.LinkText);
}
```

Этот код открывает используемый по умолчанию браузер, если он еще не был открыт, и переходит к сайту, указанному ссылкой, на которой был выполнен щелчок.

Часть приложения, связанная с редактированием, готова. Остается только реализовать загрузку и сохранение содержимого элемента управления. Для этого мы будем использовать один и тот же фиксированный файл. Дважды щелкните на кнопке Load (Загрузить) и добавьте следующий код:

```
private void buttonLoad_Click(object sender, EventArgs e)
{
   try
   {
      richTextBoxText.LoadFile("Test.rtf");
   }
   catch (System.IO.FileNotFoundException)
   {
      MessageBox.Show("No file to load yet");
   }
}
```

Вот и все! Мы больше ничего не можем сделать. Поскольку мы имеем дело с файлами, всегда существует вероятность возникновения исключений, которые должны быть обработаны. Метод `Load` выполняет обработку исключения, генерируемого, если файл не существует. Сохранение файла — столь же простая задача. Дважды щелкните на кнопке Save (Сохранить) и добавьте следующий код:

```
private void buttonSave_Click(object sender, EventArgs e)
{
   try
   {
      richTextBoxText.SaveFile("Test.rtf");
   }
   catch (System.Exception err)
   {
      MessageBox.Show(err.Message);
   }
}
```

Снова запустите пример, сформатируйте какой-либо текст и щелкните на кнопке Save. Очистите содержимое текстового поля и щелкните на кнопке Load — только что сохраненный текст должен снова появиться.

Результат должен выглядеть подобно показанному на рис. 15.10.

Рис. 15.10. *Приложение* `Simple Text Editor` *в работе*

Элементы управления ListBox и CheckedListBox

Списки используются для отображения перечней строк, из которых одновременно можно выбирать одну или более строк. Подобно флажкам и переключателям, списки предоставляют способ затребовать от пользователя выбор одного или более элементов. Список следует использовать в тех случаях, когда во время разработки неизвестно точное количество значений, из которых пользователь может производить выбор (например, список сотрудников). Даже если все возможные значения известны во время разработки, о применении списка следует подумать при наличии большого количества значений.

Класс `ListBox` является производным от класса `ListControl`, который предоставляет базовую функциональность списочных элементов управления, поставляемых в .NET Framework.

Еще один вид доступного списка — `CheckedListBox`. Будучи производным от класса `ListBox`, он предоставляет список, подобно `ListBox`, но кроме текстовых строк он поддерживает также флажки для элементов в списке.

Свойства элемента управления ListBox

Свойства, описанные в табл. 15.13, существуют в обоих классах `ListBox` и `CheckedListBox`, если только не указано иное.

Таблица 15.13. Свойства элемента управления ListBox

Свойство	Описание
`SelectedIndex`	Это значение указывает начинающийся с нуля индекс элемента, выбранного в списке. Если список поддерживает одновременный выбор нескольких элементов, это свойство содержит индекс первого элемента в выделенном фрагменте
`ColumnWidth`	Это свойство указывает ширину столбцов в списке, содержащем несколько столбцов
`Items`	Коллекция `Items` содержит все элементы списка. Свойства этой коллекции используются для добавления и удаления элементов
`MultiColumn`	Список может содержать более одного столбца. Используйте это свойство для выяснения или установки того, должны ли значения отображаться в виде столбцов
`SelectedIndices`	Коллекция, которая содержит начинающиеся с нуля индексы выбранных элементов списка
`SelectedItem`	В списке, в котором возможен выбор только одного элемента, это свойство содержит выбранный элемент, если таковой существует. В списке, в котором возможен выбор более одного элемента, оно будет содержать первый из выбранных элементов
`SelectedItems`	Коллекция, которая содержит все элементы, выбранные в текущий момент времени

Окончание табл. 15.13

Свойство	Описание
SelectionMode	Перечисление ListSelectionMode в списке позволяет выбирать один из четырех режимов выбора: None — ни один элемент не может быть выбран One — только один элемент может быть выбран в каждый конкретный момент времени Multisimple — возможен выбор нескольких элементов; при использовании этого стиля после щелчка элемент становится выбранным и остается таким даже в случае щелчка на другом элементе до повторного щелчка на нем MultiExtended — возможен выбор нескольких элементов Для выбора элементов можно использовать <Ctrl>, <Shift> и клавиши со стрелками. В отличие от режима MultiSimple, простой щелчок на одном элементе, а затем щелчок на другом будет приводить к выбору только второго элемента, на котором был выполнен щелчок
Sorted	Когда значение этого свойства установлено равным true, элемент управления ListBox упорядочивает содержащиеся в нем элементы по алфавиту
Text	Со свойствами Text мы встречались при рассмотрении многих элементов управления, но это работает иначе, чем все ранее рассмотренные. При установке свойства Text элемента управления ListBox он выполняет поиск элемента, который соответствует указанному тексту, и выбирает его. При получении свойства Text возвращенным значением является первый выбранный элемент списка. Использование этого свойства невозможно, если значение свойства SelectionMode равно None
CheckedIndices	(Только для CheckedListBox.) Это свойство — коллекция, содержащая индексы всех элементов в CheckedListBox, которые находятся в состоянии Checked или Indeterminate
CheckedItems	(Только для CheckedListBox.) Это свойство — коллекция, содержащая все элементы в CheckedListBox, которые находятся в состоянии Checked или Indeterminate
CheckOnClick	(Только для CheckedListBox.) Если значение этого свойства равно true, элемент будет изменять свое состояние при каждом щелчке на нем
ThreeDCheckBoxes	(Только для CheckedListBox.) Устанавливая это свойство, можно выбирать для использования плоские и обычные элементы управления CheckBox

Методы элемента управления ListBox

Для эффективной работы со списком необходимо знать набор методов, которые можно вызывать. Наиболее часто используемые методы описаны в табл. 15.14. Если не указано иное, эти методы принадлежат обоим классам ListBox и CheckedListBox.

Таблица 15.14. Часто используемые методы элемента управления `ListBox`

Метод	Описание
`ClearSelected()`	Очищает все выделенные фрагменты в элементе управления `ListBox`
`FindString()`	Находит в элементе управления `ListBox` первую строку, начинающуюся с указанной строки. Например, `FindString("a")` найдет в `ListBox` строку, начинающуюся с символа `a`.
`FindStringExact()`	Подобен методу `FindString`, но с указанной должна совпадать вся строка
`GetSelected()`	Возвращает значение, указывающее, выбран ли элемент
`SetSelected()`	Устанавливает или очищает выбор элемента
`ToString()`	Возвращает элемент, выбранный в текущий момент
`GetItemChecked()`	(Только для `CheckedListBox`.) Возвращает значение, указывающее, выбран ли элемент
`GetItemCheckState()`	(Только для `CheckedListBox`.) Возвращает значение, указывающее состояние установки флажка элемента
`SetItemChecked()`	(Только для `CheckedListBox`.) Устанавливает указанный элемент в состояние `Checked`
`SetItemCheckState()`	(Только для `CheckedListBox`.) Устанавливает состояние установки флажка элемента

События элемента управления `ListBox`

Обычно события, с которыми необходимо иметь дело при работе с элементами управления `ListBox` или `CheckedListBox`, связаны с выделенными пользователем элементами. События элемента управления `ListBox` описаны в табл. 15.15.

Таблица 15.15. События элемента управления `ListBox`

Событие	Описание
`ItemCheck`	(Только для `CheckedListBox`.) Происходит при изменении состояния установки флажка одного из элементов списка
`SelectedIndexChanged`	Происходит при изменении индекса выбранного элемента

В следующем практическом занятии будут созданы оба элемента управления — `ListBox` и `CheckedListBox`. Пользователи могут устанавливать флажки элементов в элементе управления `CheckedListBox`, после чего щелкать на кнопке для перемещения выбранных элементов в обычный список `ListBox`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с элементом управления `ListBox`

Создайте диалоговое окно, как описано ниже.

1. Создайте новое Windows-приложение `Lists` в каталоге `C:\BegVCSharp\chapter15`.

2. Добавьте на форму элементы управления ListBox, CheckedListBox и Button и измените их имена, как показано на рис. 15.11.
3. Измените значение свойства Text кнопки на Move.
4. Измените значение свойства CheckOnClick элемента управления CheckedListBox на true.

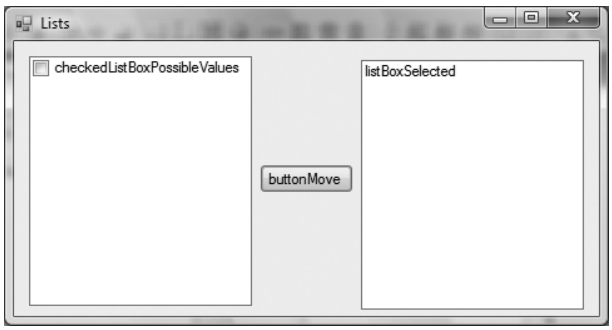


***Рис. 15.11**. Пользовательский интерфейс приложения Lists*

5. Откройте редактор элементов для элемента управления checkedListBox, щелкнув на кнопке со значком многоточия (...). Затем в отдельных строках введите One, Two, Three, Four, Five, Six, Seven, Eight и Nine и щелкните на кнопке OK.
6. Вставьте в CheckedListBox еще один элемент, но сделайте это из кода, как показано в следующем фрагменте.

```
public Form1()
{
   InitializeComponent();
   checkedListBoxPossibleValues.Items.Add("Ten");
}
```

*Фрагмент кода **Chapter15\TextBoxControls\Form1.cs***

Теперь пора заняться обработчиками событий. Добавьте код для перемещения элементов из CheckedListBox в обычный список ListBox. Когда пользователь щелкает на кнопке Move (Переместить), нужно найти отмеченные флажками элементы и скопировать их в список, расположенный справа.

7. Дважды щелкните на кнопке buttonMove и введите следующий код:

```
private void buttonMove_Click(object sender, EventArgs e)
  {
     if   (checkedListBoxPossibleValues.CheckedItems.Count > 0)
     {
        listBoxSelected.Items.Clear();
        foreach (string item in checkedListBoxPossibleValues.CheckedItems)
        {
           listBoxSelected.Items.Add(item.ToString());
        }
        for (int i = 0; i < checkedListBoxPossibleValues.Items.Count; i++)
           checkedListBoxPossibleValues.SetItemChecked(i, false);
     }
  }
```

Описание работы

Работа начинается с проверки свойства `Count` коллекции `CheckedItems`. Его значение будет больше нуля, если любые элементы в коллекции выбраны. Затем мы очищаем все элементы в списке `listBoxSelected` и проходим в цикле по коллекции `CheckedItems`, добавляя каждый элемент в список `listBoxSelected`. И, наконец, мы удаляем все флажки в `CheckedListBox`.

Теперь во время работы приложения можно будет выбирать элементы в списке слева и, щелкая на кнопке Move, добавлять эти же элементы в список справа. На этом рассмотрение элементов управления `ListBox` завершено, и мы переходим к несколько более интересной теме — элементам управления `ListView`.

Элемент управления `ListView`

На рис. 15.12 показан, вероятно, наиболее известный элемент управления `ListView` в среде Windows. Несмотря на то что теперь Windows предлагает множество дополнительных возможностей отображения файлов и папок, вы, несомненно, узнаете некоторые возможности, предоставляемые в элементе управления `ListView`, такие как отображение крупных значков, представление подробных сведений и т.п.

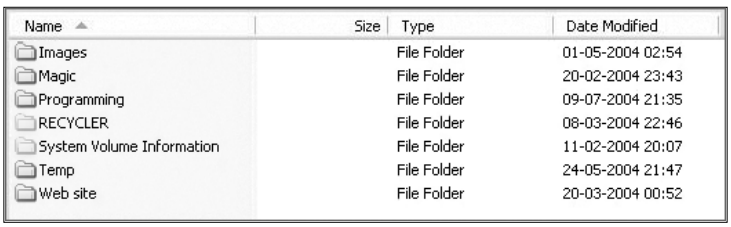


***Рис. 15.12**. Пример элемента управления `ListView`*

Представление в виде списка обычно используется для данных, применительно к которым пользователь располагает определенным контролем над степенью подробности и стилем их представления. Данные, содержащиеся в элементе управления, можно представлять в виде столбцов и строк, подобно таблице, в виде единственного столба или в виде различных значков. Наиболее часто используемое представление в виде списка (см. рис. 15.12) служит для навигации по папкам файловой системы.

Элемент управления `ListView`, пожалуй, наиболее сложный из всех, рассмотренных в этой главе, и освещение всех его аспектов выходит за рамки настоящей книги. В этой главе представлен пример, в котором используются многие важные функциональные возможности элемента управления `ListView`, а также дано подробное описание многочисленных доступных свойств, событий и методов, что все вместе обеспечит прочный фундамент для дальнейшей работы. Кроме того, будет рассмотрен элемент управления `ImageList`, который предназначен для хранения изображений, используемых в элементе управления `ListView`.

Свойства элемента управления `ListView`

Свойства элемента управления `ListView` описаны в табл. 15.16.

Таблица 15.16. Свойства элемента управления `ListView`

Свойство	Описание
`Activation`	Управляет способом активизации элемента пользователем в списочном представлении. Возможные значения следующие: `Standard` — эта настройка отражает способ активизации, выбранный пользователем для своего компьютера `OneClick` — одиночный щелчок на элементе активизирует его `TwoClick` — двойной щелчок на элементе активизирует его

`Alignment`	Управляет способом выравнивания элементов в представлении в виде списка. Четыре возможных значения перечислены ниже:
	`Default` — если пользователь перетаскивает элемент, тот остается там, где был оставлен
	`Left` — элементы выравниваются по левому краю элемента управления `ListView`
	`Top` — элементы выравниваются по верхнему краю элемента управления `ListView`
	`SnapToGrid` — элемент управления содержит невидимую сетку, к которой будут привязаны элементы

`AllowColumnReorder`	Если значение этого свойства установлено в `true`, пользователь имеет возможность изменять порядок следования столбцов в списочном представлении. При использовании этой возможности следует удостовериться, что подпрограммы, которые заполняют списочное представление, способны правильно вставлять элементы даже после изменения порядка следования столбцов
`AutoArrange`	Если значение этого свойства установлено в `true`, элементы будут автоматически выстраиваться в соответствии со свойством `Alignment`. Если пользователь перетаскивает элемент в центр списочного представления, а значение свойства `Alignment` равно `Left`, элемент автоматически переместится к левому краю представления. Это свойство имеет смысл только в том случае, если значение свойства `View` расно `LargeIcon` или `SmallIcon`
`CheckBoxes`	Если значение этого свойства установлено в `true`, слева от каждого элемента в списочном представлении будет отображаться элемент управления `CheckBox`. Это свойство имеет смысл только в том случае, если значением свойства `View` является `Details` или `List`
`CheckedIndices` `CheckedItems`	Предоставляют доступ, соответственно, к коллекциям индексов и элементов, помеченных флажками
`Columns`	Представление в виде списка может содержать столбцы. Это свойство предоставляет доступ к коллекции столбцов, посредством которой можно добавлять или удалять столбцы
`FocusedItem`	Содержит элемент, обладающий фокусом в списке. Если ни один элемент не выбран, это значение равно `null`
`FullRowSelect`	Когда значение этого свойства равно `true`, и на элементе выполнен щелчок, вся строка, содержащая этот элемент, выделяется. Если значением является `false`, выделяется только сам элемент

Продолжение табл. 15.16

Свойство	Описание
GridLines	Если значение этого свойства установлено в true, в списочном представлении рисуются линии сетки между строками и столбцами. Это свойство имеет смысл только в том случае, если значением свойства View является Details
HeaderStyle	Управляет способом отображения заголовков столбцов. Существуют три стиля: Clickable — заголовок столбца работает подобно кнопке NonClickable — заголовки столбцов не реагируют на щелчки кнопкой мыши None — заголовки столбцов не отображаются
HoverSelection	Когда значение этого свойства равно true, пользователь может выбирать элемент в списочном представлении, задерживая указатель мыши над ним
Items	Коллекция элементов в представлении в виде списка
LabelEdit	Когда значение этого свойства равно true, пользователь может редактировать содержимое первого столбца в представлении Details (Подробные сведения)
LabelWrap	Если значение этого свойства равно true, метки будут занимать столько строк, сколько требуется для отображения всего текста
LargeImageList	Содержит объект ImageList, хранящий крупные изображения. Эти изображения могут использоваться, когда значение свойства View равно LargeIcon
MultiSelect	Установка этого свойства в true позволяет пользователю выбирать несколько элементов
Scrollable	Установка этого свойства в true приводит к отображению линеек прокрутки
SelectedIndices SelectedItems	Содержит коллекции, которые, соответственно, хранят выбранные индексы и элементы
SmallImageList	Когда значение свойства View равно SmallIcon, это свойство содержит объект ImageList, хранящий используемые изображения
Sorting	Позволяет представлению в виде списка выполнять упорядочение элементов, которые оно содержит. Существует три возможных режима: Ascending (по возрастанию), Descending (по убыванию) и None (без сортировки).
StateImageList	ImageList содержит маски изображений, которые накладываются на изображения LargeImageList и SmallImageList для представления нестандартных состояний
TopItem	Возвращает элемент, расположенный в верхней части представления в виде списка

Окончание табл. 15.16

Свойство	Описание
`View`	Представление в виде списка может отображать свои элементы в следующих основных режимах: `LargeIcon` — все элементы представляются в виде крупного значка (32×32) и метки `SmallIcon` — все элементы представляются в виде маленького значка (16×16) и метки `List` — отображается только один столбец, который может содержать значок и метку `Details` — возможно отображение любого количества столбцов, но только первый столбец может содержать значок `Tile` (доступно только на платформах Windows XP и последующих версий Windows) — отображает крупный значок, а справа от него — метку и информацию об подэлементе

Методы элементы управления `ListView`

Для столь сложного элемента, как `ListView`, определено удивительно мало специализированных методов (табл. 15.17).

Таблица 15.17. Методы элемента управления `ListView`

Метод	Описание
`BeginUpdate()`	Указывает представлению в виде списка о необходимости прекращения прорисовки обновлений до момента вызова метода `EndUpdate()`. Этот метод полезен при одновременной вставке множества элементов, поскольку он предотвращает мерцание представления и радикально увеличивает быстродействие
`Clear()`	Полностью очищает представление в виде списка. Все элементы и столбцы удаляются
`EndUpdate()`	Этот метод вызывают после вызова метода `BeginUpdate`. При его вызове представление в виде списка прорисовывает все свои элементы
`EnsureVisible()`	Указывает представлению в виде списка о необходимости выполнить прокрутки для отображения элемента с указанным индексом
`GetItemAt()`	Возвращает объект `ListViewItem` в позиции `x`, `y` представления в виде списка

События элемента управления `ListView`

События элемента управления `ListView` описаны в табл. 15.18.

Класс `ListViewItem`

Элемент в списочном представлении всегда является экземпляром класса `ListViewItem`. Этот класс содержит такую информацию, как текст и индекс значка, которую нужно отображать. Объекты `ListViewItem` имеют свойство `SubItems`, которое содержит экземпляры еще одного класса — `ListViewSubItem`.

Таблица 15.18. Часто используемые события элемента управления `ListView`

Событие	Описание
`AfterLabelEdit`	Происходит после редактирования метки
`BeforeLabelEdit`	Происходит до того, как пользователь начинает редактировать метку
`ColumnClick`	Происходит при щелчке на столбце
`ItemActivate`	Происходит при активизации элемента

Эти подэлементы отображаются, если элемент управления `ListView` находится в режиме `Details` или `Tile`. Каждый подэлемент представляет столбец в списочном представлении. Главное отличие между подэлементами и основными элементами состоит в том, что подэлемент не может отображать значок.

Объекты `ListViewItem` добавляются в `ListView` через коллекцию `Items`, а объекты `ListViewSubItems` в `ListViewItem` — посредством коллекции `SubItems` в объекте `ListViewItem`.

Класс `ColumnHeader`

Чтобы списочное представление отображало заголовки столбцов, в коллекцию `Columns` объекта `ListView` добавляют экземпляры класса `ColumnHeader`. Этот класс предоставляет заголовок для столбцов, который может отображаться, когда элемент управления `ListView` находится в режиме `Details`.

Элемент управления `ImageList`

Элемент управления `ImageList` предоставляет коллекцию, которую можно применять для хранения изображений, используемых в других элементах управления формы. В списке изображений можно хранить изображения любого размера, но внутри каждого элемента управления все изображения должны быть одного размера. Применительно к элементу управления `ListView` это означает, что для отображения и крупных, и мелких изображений требуются два элемента управления `ImageList`.

Класс `ImageList` — первый из представленных в этой главе элементов управления, который сам по себе невидим во время выполнения. При его перетаскивании на разрабатываемую форму, он помещается не в саму форму, а в лоток под ней, содержащий все аналогичные компоненты. Эта замечательная особенность призвана предотвращать загромождение рабочей области визуального конструктора форм элементами управления, которые не являются частью интерфейса пользователя. Манипулирование этим элементом управления осуществляется точно так же, как и любым другим, за исключением того, что его нельзя перемещать поверх формы.

Изображения в элемент управления `ImageList` можно добавлять как во время разработки, так и во время выполнения. Если во время разработки изображения, которые нужно отображать, известны, их можно добавить, щелкая на кнопке, расположенной справа от свойства `Images`. В результате откроется диалоговое окно, где можно перейти к изображениям, которые нужно вставить. Добавление изображений во время выполнения осуществляется через коллекцию `Images`.

Ознакомиться с использованием элемента управления `ListView` и связанных с ним списков изображений лучше всего на примере. В следующем практическом занятии будет создано диалоговое окно со списочным представлением и двумя списками изображений. Списочное представление будет отображать файлы и папки, хранящиеся на жестком диске. Для простоты мы не будем извлекать соответствующие значки из файлов и папок, а будем использовать стандартный значок папки для папок и значок текстового файла для файлов.

Двойной щелчок на папке позволяет перейти к дереву папки, а кнопка Back (Назад) дает возможность перемещаться вверх по дереву. Пять переключателей служат для изменения режима представления в виде списка во время выполнения. Двойной щелчок на файле будет приводить к его запуску.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с элементом управления `ListView`

Как всегда, начнем с создания интерфейса пользователя.

1. Создайте новое Windows-приложение `ListView` в каталоге `C:\BegVCSharp\Chapter15`.
2. Добавьте на форму элементы управления `ListView`, `Button`, `Label` и `GroupBox`. Затем добавьте в рамку `GroupBox` пять переключателей. Форма должна выглядеть, как показано на рис. 15.13. Чтобы установить ширину элемента управления `Label`, установите значение его свойства `AutoSize` в `false`. Сделайте ширину элемента управления `Label` равной ширине элемента управления `ListView`.

***Рис. 15.13**. Пользовательский интерфейс приложения `ListView`*

3. Назначьте имена элементам управления в соответствии с рис. 15.13. Элемент `ListView` не будет отображать свое имя, как показано на рисунке. Дополнительный элемент был добавлен лишь для того, чтобы отобразить имя в этом примере — его добавление не обязательно.
4. Измените свойство `Text` переключателей и кнопки, чтобы оно совпадало с именами, за исключением наименования элементов управления, а в качестве свойства `Text` формы установите `ListView`.
5. Очистите свойство `Text` метки.
6. Добавьте на форму два элемента управления `ImageList`, дважды щелкнув на значке этого элемента в разделе Components (Компоненты) панели инструментов. Переименуйте элементы управления на `imageListSmall` и `imageListLarge`.
7. Измените значение свойства `Size` элемента управления `ImageList` по имени `imageListLarge` на `32, 32`.
8. Щелкните на кнопке, расположенной справа от свойства `Images` списка изображений `imageListLarge`, чтобы открыть диалоговое окно, где можно будет перейти к изображениям, которые требуется вставить.

9. Щелкните на кнопке Add (Добавить) и перейдите к папке ListView в каталоге кода для этой главы. Нужные файлы — Folder 32x32.ico и Text 32x32.ico.
10. Удостоверьтесь, что значок папки находится в верхней части списка.
11. Повторите шаги 8 и 9 для компонента imageListSmall, выбирая версии значков с размерами 16×16.
12. Установите значение свойства Checked переключателя radioButtonDetails в true.
13. Установите свойства списочного представления, как указано в следующей таблице:

Свойство	Значение
LargelmageList	imageListLarge
SmalllmageList	imageListSmall
View	Details

Добавление обработчиков событий

Мы завершили создание интерфейса пользователя и теперь можем переходить к написанию кода. Первым делом необходимо поле для хранения папок, по которым выполняется навигация, чтобы к ним можно было возвращаться по щелчку на кнопке Back. Поскольку будут сохраняться абсолютные пути к папкам, для этого выбирается коллекция StringCollection:

```
partial class Form1: Form
{
  private System.Collections.Specialized.StringCollection folderCol;
```

Мы не создали ни одного заголовка столбцов в визуальном конструкторе форм, поэтому их нужно создать теперь с помощью метода CreateHeadersAndFillListView():

```
private void CreateHeadersAndFillListView()
{
  ColumnHeader colHead;

  colHead = new ColumnHeader();
  colHead.Text = "Filename";
  listViewFilesAndFolders.Columns.Add(colHead); // Вставка заголовка

  colHead = new ColumnHeader();
  colHead.Text = "Size";
  listViewFilesAndFolders.Columns.Add(colHead); // Вставка заголовка

  colHead = new ColumnHeader();
  colHead.Text = "Last accessed";
  listViewFilesAndFolders.Columns.Add(colHead); // Вставка заголовка
}
```

*Фрагмент кода **Chapter15\ListView\Form1.cs***

Мы начинаем с объявления единственной переменной colHead, используемой для создания трех заголовков столбцов. Для каждого из трех заголовков мы создаем переменную заново и присваиваем ей Text, прежде чем добавить ее в коллекцию Columns элемента управления ListView.

Заключительное действие по инициализации формы, поскольку она отображается впервые, сводится к заполнению списочного представления файлами и папками, загруженными с жесткого диска. Для этого служит другой метод:

```
private void PaintListView(string root)
{
  try
  {
    ListViewItem lvi;
    ListViewItem.ListViewSubItem lvsi;

    if (string.IsNullOrEmpty(root))
      return;

    DirectoryInfo dir = new DirectoryInfo(root);
    DirectoryInfo[] dirs = dir.GetDirectories();
    FileInfo[] files = dir.GetFiles();

    listViewFilesAndFolders.Items.Clear();
    labelCurrentPath.Text = root;
    listViewFilesAndFolders.BeginUpdate();

    foreach (DirectoryInfo di in dirs)
    {
      lvi = new ListViewItem();
      lvi.Text = di.Name;
      lvi.ImageIndex = 0;
      lvi.Tag = di.FullName;

      lvsi = new ListViewItem.ListViewSubItem();
      lvsi.Text = "";
      lvi.SubItems.Add(lvsi);

      lvsi = new ListViewItem.ListViewSubItem();
      lvsi.Text = di.LastAccessTime.ToString();
      lvi.SubItems.Add(lvsi);
      listViewFilesAndFolders.Items.Add(lvi);
    }

    foreach (FileInfo fi in files)
    {
      lvi = new ListViewItem();
      lvi.Text = fi.Name;
      lvi.ImageIndex = 1;
      lvi.Tag = fi.FullName;

      lvsi = new ListViewItem.ListViewSubItem();
      lvsi.Text = fi.Length.ToString();
      lvi.SubItems.Add(lvsi);

      lvsi = new ListViewItem.ListViewSubItem();
      lvsi.Text = fi.LastAccessTime.ToString();
      lvi.SubItems.Add(lvsi);

      listViewFilesAndFolders.Items.Add(lvi);
    }
    listViewFilesAndFolders.EndUpdate();
  }
  catch (System.Exception err)
  {
    MessageBox.Show("Error: " + err.Message); // возникла ошибка
  }
}
```

Фрагмент кода ***Chapter15\TextBoxControls\Form1.cs***

Описание работы

Перед первым из двух блоков `foreach` мы вызываем метод `BeginUpdate()` на элементе управления `ListView`. Помните, что метод `BeginUpdate()` элемента управления `ListView` указывает ему о необходимости прекратить обновление видимой области до момента вызова метода `EndUpdate()`. Отказ от вызова этого метода привел бы к замедлению заполнения списочного представления и возможному мерцанию при добавлении элементов. Сразу после второго блока `foreach` мы вызываем метод `EndUpdate()`, который вынуждает элемент управления `ListView` отрисовать элементы, которыми он был заполнен.

Оба блока `foreach` содержат интересный код. Мы начинаем с создания нового экземпляра `ListViewItem`, а затем устанавливаем свойство `Text` в соответствии с именем файла или папки, которую собираемся вставить. Свойство `ImageIndex` объекта `ListViewItem` ссылается на индекс элемента в одном из списков изображений. Поэтому важно, чтобы значки имели одинаковые индексы в обоих списках изображений. Для сохранения полностью определенного пути к папкам и файлам используется свойство `Tag`, которое будет задействовано, когда пользователь дважды щелкнет на элементе.

Затем мы создаем два подэлемента. Им просто присваивается текст, предназначенный для отображения, после чего они добавляются в коллекцию `SubItems` компонента `ListViewItem`.

И, наконец, `ListViewItem`, добавляется в коллекцию `Items` элемента управления `ListView`. Элемент управления `ListView` достаточно интеллектуален, чтобы просто игнорировать подэлементы, если режим просмотра отличается от `Details`, поэтому мы добавляем подэлементы, независимо от текущего режима просмотра.

Обратите внимание, что некоторые аспекты кода остались без внимания, а именно — строки кода, в которых действительно получается информация о файлах:

```
// Получение информации о корневой папке.
DirectoryInfo dir = new DirectoryInfo(root);

// Извлечение файлов и папок из корневой папки.
DirectoryInfo[] dirs = dir.GetDirectories();
FileInfo[] files = dir.GetFiles();
```

В приведенных строках кода для доступа к файлам применяются классы из пространства имен `System.IO`, поэтому в верхнюю часть кода нужно добавить следующий раздел `using`:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
```

Более подробно доступ к файлам и пространство имен `System.IO` рассматриваются в главе 21. Однако чтобы можно было понять происходящее, отметим, что метод `GetDirectories()` объекта `DirectoryInfo` возвращает коллекцию объектов, которые представляют папки в просматриваемом каталоге, а метод `GetFiles()` — коллекцию объектов, представляющих файлы в текущем каталоге. Эти коллекции можно проходить циклом, как это только что было сделано, используя свойство `Name` объекта для возврата имени соответствующего каталога или файла и создания объекта `ListViewItem` для хранения этой строки.

Чтобы элемент управления `ListView` отобразил корневую папку, остается только вызвать две функции в конструкторе формы. Одновременно создается экземпляр коллекции `StringCollection` по имени `folderCol` с корневой папкой:

```
InitializeComponent();

folderCol = new System.Collections.Specialized.StringCollection();
CreateHeadersAndFillListView();
PaintListView(S"C:\");
folderCol.Add(@"C:\");
```

Чтобы пользователи имели возможность дважды щелкать на элементе в компоненте ListView для просмотра папок, необходимо подписаться на событие ItemActivate. Выберите ListView в окне визуального конструктора и дважды щелкните на событии ItemActivate в списке Events окна Properties.

Соответствующий обработчик события имеет следующий вид:

```
private void listViewFilesAndFolders_ItemActivate(object sender, EventArgs e)
{
   System.Windows.Forms.ListView lw = (System.Windows.Forms.ListView)sender;
   string filename = lw.SelectedItems[0].Tag.ToString();
   if (lw.SelectedItems[0].ImageIndex != 0)
   {
      try
      {
         System.Diagnostics.Process.Start(filename);
      }
      catch { return; }
   }
   else
   {
      PaintListView(filename);
      folderCol.Add(filename);
   }
}
```

Свойство Tag выбранного элемента содержит полностью определенный путь к файлу или папке, где был выполнен двойной щелчок. Известно, что изображение с индексом 0 — это папка, поэтому по данному индексу можно определить, является элемент файлом или папкой. Если он — файл, предпринимается попытка загрузки файла. Если он — папка, вызывается метод PaintListView() с новой папкой, а затем новая папка добавляется в коллекцию folderCol.

Прежде чем переходить к переключателям, нужно завершить построение возможностей просмотра, добавив событие Click в кнопку Back. Дважды щелкните на кнопке и поместите в обработчик событий следующий код:

```
private void buttonBack_Click(object sender, EventArgs e)
{
   if (folderCol.Count > 1)
   {
      PaintListView(folderCol[folderCol.Count - 2].ToString());
      folderCol.RemoveAt(folderCol.Count - 1);
   }
   else
      PaintListView(folderCol[0].ToString());
}
```

Если коллекция folderCol содержит больше одного элемента, значит, мы находимся не в корневом каталоге браузера и нужно вызвать метод PaintListView() с путем к предыдущей папке. Последним элементом в коллекции folderCol является текущая папка. Поэтому необходимо извлечь предпоследний элемент. Затем мы удаляем последний элемент коллекции и делаем новый последний элемент текущей папкой. Если коллекция содержит только один элемент, мы просто вызываем метод PaintListView() с этим элементом.

Теперь остается только обеспечить возможность изменения типа просмотра для элемента управления `ListView`. Дважды щелкните на каждом переключателе и добавьте следующий код:

```
private void radioButtonLargeIcon_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rdb = (RadioButton)sender;
  if (rdb.Checked)
    this.listViewFilesAndFolders.View = View.LargeIcon;
}
private void radioButtonSmallIcon_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rdb = (RadioButton)sender;
  if (rdb.Checked)
    listViewFilesAndFolders.View = View.SmallIcon;
}
private void radioButtonList_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rdb = (RadioButton)sender;
  if (rdb.Checked)
    listViewFilesAndFolders.View = View.List;
}
private void radioButtonDetails_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rdb = (RadioButton)sender;
  if (rdb.Checked)
    listViewFilesAndFolders.View = View.Details;
}
private void radioButtonTile_CheckedChanged(object sender, EventArgs e)
{
  RadioButton rdb = (RadioButton)sender;
  if (rdb.Checked)
    listViewFilesAndFolders.View = View.Tile;
}
```

Фрагмент кода ***Cpapter15\ListView\Form1.cs***

Здесь осуществляется проверка переключателя с целью выяснения, изменилось ли его состояние на `Checked`; если это так, свойство `View` элемента управления `ListView` соответствующим образом устанавливается.

На этом пример применения элемента управления `ListView` завершен. Результат его запуска должен выглядеть, как показано на рис. 15.14.

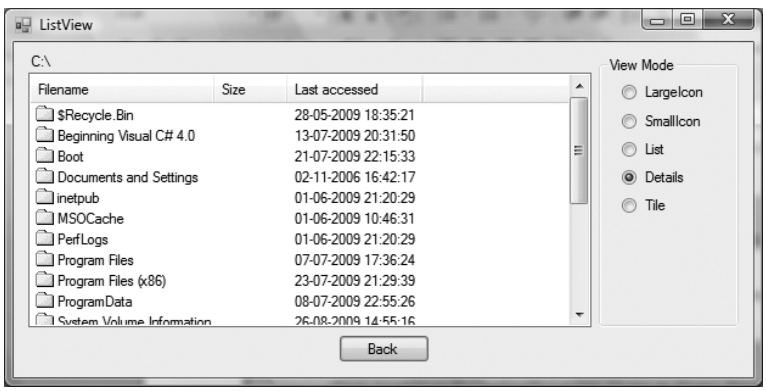


Рис. 15.14*. Приложение* `ListView` *в работе*

Элемент управления TabControl

Элемент управления TabControl предоставляет простой способ организации диалоговых окон в логические части, доступные посредством вкладок в верхней части элемента управления. TabControl содержит элементы TabPage, которые работают подобно элементу управления GroupBox, поскольку группируют элементы управления, хотя и являются несколько более сложными.

Использование элемента управления TabControl не представляет сложности. Вы просто добавляете нужное количество вкладок для отображения коллекции объектов TabPage элемента управления, а затем перетаскиваете элементы управления, которые требуется отображать, на соответствующие страницы.

Свойства элемента управления TabControl

Свойства объекта TabControl (описанные в табл. 15.19) в основном служат для управления внешним видом контейнера объектов TabPage — в частности, отображаемых вкладок.

Таблица 15.19. Свойства элемента управления TabControl

Свойство	Описание
Alignment	Это свойство управляет местом отображения вкладок элемента управления TabControl. По умолчанию они отображаются в верхней части элемента управления
Appearance	Свойство Appearance управляет способом отображения вкладок. Вкладки могут отображаться как обычные кнопки или плоскими
HotTrack	Если значение этого свойства установлено в true, внешний вид вкладок элемента управления изменяется при прохождении над ними указателя мыши
Multiline	Если значение этого свойства установлено в true, возможно наличие нескольких рядов вкладок
RowCount	RowCount возвращает количество рядов отображенных в текущий момент вкладок
SelectedIndex	Это свойство возвращает или устанавливает индекс выбранной вкладки
SelectedTab	SelectedTab возвращает или устанавливает выбранную вкладку. Обратите внимание, что это свойство работает с фактическими экземплярами объектов TabPage
TabCount	TabCount возвращает общее количество вкладок
TabPages	Это свойство — коллекция объектов TabPage в элементе управления. Данная коллекция используется для добавления и удаления объектов TabPage

Работа с элементом управления TabControl

Элемент TabControl работает несколько иначе, чем остальные элементы управления, рассмотренные до сих пор. Фактически этот элемент управления представляет собой контейнер для страниц вкладок, служащий для отображения страниц. При двойном щелчке на TabControl в панели инструментов создается элемент управления, который уже содержит два элемента TabPage.

При выборе элемента управления в его верхнем правом углу появляется маленькая кнопка с изображением треугольника. Щелчок на этой кнопке ведет к разворачиванию

небольшого окна. Это окно Actions Window (Окно действий) позволяет легко получать доступ к выбранным свойствам и методам элемента управления. Вы могли обратить на него внимание раньше, поскольку многие элементы управления в Visual Studio обладают этой функциональной возможностью, но `TabControl` — первый из элементов управления, описанных в этой главе, который действительно позволяет выполнить нечто интересное в окне Actions Window. Упомянутое окно элемента управления `TabControl` позволяет легко добавлять и удалять элементы `TabPage` на этапе проектирования.

Процедура добавления вкладок к элементу управления `TabControl`, описанная в предыдущем абзаце, призвана обеспечить быстрое создание и начало работы с элементом управления. Однако если требуется изменить поведение или стиль вкладок, следует использовать диалоговое окно `TabPages`, доступное посредством кнопки при выборе свойства `TabPages` в окне Properties. Свойство `TabPages` является также коллекцией, которая применяется для доступа к отдельным страницам в элементе управления `TabControl`.

Как только нужные элементы `TabPage` добавлены, элементы управления можно добавлять к страницам так же, как это делалось раньше при работе с элементом управления `GroupBox`. В следующем практическом занятии демонстрируются основы работы с этим элементом управления.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с элементом управления `TabControl`

Выполните перечисленные ниже действия, чтобы создать Windows-приложение, которое демонстрирует разработку элементов управления, расположенных на различных страницах элемента `TabControl`.

1. Создайте новое Windows-приложение `TabControl` в каталоге `C:\BegVCSharp\Chapter15`.
2. Перетащите элемент управления `TabControl` из панели инструментов Toolbox на форму. Как и `GroupBox`, элемент `TabControl` находится на вкладке Containers (Контейнеры) панели Toolbox.
3. Найдите свойство `TabPages` и щелкните на расположенной справа от него кнопке, чтобы открыть диалоговое окно TabPage Collection Editor (Редактор коллекции `TabPage`).
4. Измените свойство `Text` страниц вкладок соответственно на `Tab one` и `Tab two` и щелкните на кнопке OK, чтобы закрыть диалоговое окно.
5. Страницы вкладок для работы можно выбирать, щелкая на вкладках в верхней части элемента управления. Выберите вкладку Tab one. Перетащите кнопку на элемент управления. Удостоверьтесь, что кнопка помещена внутрь рамки элемента управления `TabControl`. В случае ее помещения снаружи этой рамки кнопка будет помещена на форму, а не в элемент управления.
6. Измените имя кнопки на `buttonShowMessage`, а ее свойство `Text` — на `Show Message`.
7. Щелкните на вкладке, свойство `Text` которой равно `Tab two`. Перетащите элемент управления `TextBox` на поверхность `TabControl`. Назначьте имя этому элементу управления.
8. Две созданные вкладки должны выглядеть, как показано на рис. 15.15 и 15.16.

Теперь можно обращаться к элементам управления. Если запустить код в его нынешнем состоянии, страницы вкладок отобразятся надлежащим образом.

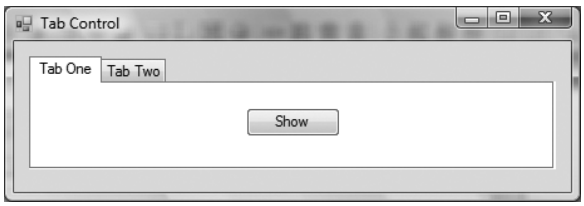


***Рис. 15.15**. Первая вкладка элемента `TabControl`*

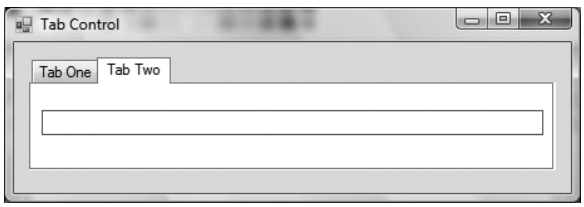


***Рис. 15.16**. Вторая вкладка элемента `TabControl`*

Для демонстрации всех возможностей элемента управления `TabControl` остается только добавить небольшой фрагмент кода, чтобы щелчок на кнопке Show Message (Показать сообщение) на одной вкладке приводил к отображению в окне сообщения текста, введенного на другой вкладке. Добавьте обработчик события `Click`, дважды щелкнув на кнопке Show Message на первой вкладке и добавив следующий код:

```
private void buttonShowMessage_Click(object sender, EventArgs e)
{
   MessageBox.Show(textBoxMessage.Text);
}
```

Описание работы

Доступ к элементу управления на вкладке производится так же, как к любому другому элементу управления в форме. Приложение извлекает свойство `Text` элемента управления `TextBox` и отображает его в окне сообщения.

Ранее в этой главе было показано, что на форме одновременно может быть выбран только один из переключателей (если только они не помещены в групповую рамку). Элементы управления `TabPage` работают точно так же, как групповые рамки, поэтому можно использовать несколько наборов переключателей на различных вкладках, обходясь без групповых рамок. Кроме того, как было показано при рассмотрении метода `buttonShowMessage_Click`, можно обращаться к элементам управления, расположенным на вкладках, отличных от той, где находится текущий элемент управления.

Последнее, что нужно знать для успешной работы с элементом управления `TabControl` — способ выяснения, какая вкладка отображается в данный момент. Для этого служат два свойства: `SelectedTab` и `SelectedIndex`. Свойство `SelectedTab` возвращает объект `TabPage` или значение `null`, если ни одной вкладки не выбрано, а свойство `SelectedIndex` — индекс выбранной вкладки или `-1`, если вкладка не выбрана. Для экспериментирования с этими свойствами выполните упражнение 2 в конце этой главы.

Резюме

В этой главе рассматривались некоторые элементы управления, которые наиболее часто используются при создании Windows-приложений, и было показано, как их применять для построения простых, но предлагающих большие возможности интерфейсов пользова-

теля. В главе были освещены свойства и события этих элементов управления, были приведены примеры, демонстрирующие их использование, и пояснено добавление обработчиков событий для конкретных событий элемента управления.

В следующей главе мы рассмотрим некоторые более сложные элементы управления и средства для создания приложений Windows Forms.

Упражнения

1. В предшествующих версиях Visual Studio было достаточно трудно заставить приложения отображать их элементы управления в стиле текущей версии Windows. Выясните, где именно в приложении Windows Forms осуществляется активизация визуальных стилей в новом проекте Windows Forms. Поэкспериментируйте с активизацией и отключением стилей и выясните, как они влияют на элементы управления в формах.
2. Измените пример приложения `TabControl`, добавив в него несколько вкладок и отобразив окно сообщения со следующим текстом: `Вы изменили текущую вкладку с <Текст только что оставленной вкладки> на <Текст текущей вкладки>`.
3. В примере применения `ListView` для сохранения полностью определенного пути к папкам и файлам в элементе управления `ListView` применялось свойство `Tag`. Измените это поведение, создав новый класс, производный от `ListViewItem`, и используйте экземпляры этого нового класса в качестве элементов в элементе управления `ListView`. Обеспечьте хранение информации о файлах и папках в новом классе с помощью свойства `FullyQualifiedPath`.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Метки	Используйте элементы управления `Label` и `LinkLabel` для отображения информации пользователям.
Кнопки	Используйте элемент управления `Button` и соответствующее событие `Click`, чтобы предоставить пользователям возможность указывать приложению о необходимости выполнения того или иного действия.
Текстовые поля	Используйте элементы управления `TextBox` и `RichTextBox` для предоставления пользователям возможности ввода простого либо форматированного текста.
Элементы управления выбором	Помните о различиях между элементами управления `CheckBox` и `RadioButton` и способах их применения. Эти элементы управления можно также группировать с помощью элемента управления `GroupBox`.
Списки	Используйте элемент управления `CheckedListBox` для предоставления списков, из которых пользователь может выбирать элементы, устанавливая флажки. Вы узнали также, как применять более распространенный элемент управления `ListBox` для построения списка, который аналогичен списку `CheckedListBox`, но не содержит флажков.
Элементы управления `ListView`	Используйте элементы управления `ListView` и `ImageList` для предоставления списка, которые пользователи могут просматривать множеством различных способов.
Элементы управления `TabControl`	Используйте элемент управления `TabControl` для группирования элементов управления на различных страницах одной формы, чтобы пользователь мог их выбирать по собственному желанию.

16

Расширенные средства Windows Forms

В ЭТОЙ ГЛАВЕ...

- Использование трех обычных элементов управления для создания изысканных меню, панелей инструментов и строк состояния
- Создание приложений MDI
- Создание собственных элементов управления

В предыдущей главе рассматривались элементы управления, которые наиболее часто используются при разработке Windows-приложений. С помощью таких элементов управления можно создавать впечатляющие диалоговые окна. Однако очень мало полноценных Windows-приложений обладают интерфейсом пользователя, который состоит из единственного диалогового окна. Чаще в этих приложениях применяется однодокументный (Single Document Interface — SDI) или многодокументный (Multiple Document Interface — MDI) интерфейс. Обычно приложения любого из этих типов интенсивно используют меню и панели инструментов, которые в предыдущей главе не рассматривались. Исправим это упущение.

Добавление технологии Windows Presentation Foundation в .NET Framework привело к появлению нескольких новых типов Windows-приложений. Они подробно рассматриваются в главе 25.

Эта глава начинается с того, на чем мы остановились при рассмотрении элементов управления в предыдущей главе. Вначале мы рассмотрим элемент управления меню, а затем перейдем к панелям инструментов и научимся связывать кнопки панелей инструментов с элементами меню и наоборот. Затем мы приступим к созданию приложений SDI и MDI, уделив основное внимание приложениям MDI, т.к., по сути, приложения SDI являются подмножеством приложений MDI.

До сих пор мы применяли только те элементы управления, которые поставляются в составе .NET Framework. Как вы убедились, эти элементы управления предоставляют широкий спектр функциональности, но в ряде случаев их оказывается недостаточно. В этих случаях можно создавать собственные элементы управления, и ближе к концу этой главы будет показано, как это делается.

Меню и панели инструментов

Сколько вы можете вспомнить Windows-приложений, которые не содержат меню или панели инструментов того или иного вида? Ни одного, не так ли? Меню и панели инструментов наверняка будут важными составными частями любого создаваемого Windows-приложения. Для облегчения их создания в пользовательских приложениях среда Visual Studio 2010 предлагает два элемента управления, которые позволяют достаточно легко создавать меню и панели инструментов, выглядящие подобно меню в самой среде Visual Studio.

Два как один

Два элемента управления, которые рассматриваются ниже, впервые появились в Visual Studio 2005 и стали существенным расширением возможностей как для разработчика-любителя, так и для профессионала. Построение приложений с профессионально выглядящими панелями инструментов и меню обычно оставалось уделом тех, кто было готов потратить значительное время для создания нестандартных инструментов рисования и тех, кто приобретал компоненты от независимых разработчиков. Создание того, что раньше могло занимать недели, теперь превратилось в простую задачу, которая может быть решена буквально за считанные секунды.

Элементы управления, которые мы будем использовать, могут быть сгруппированы в семейство элементов, имена которых заканчиваются суффиксом `Strip`. Это элементы управления `ToolStrip`, `MenuStrip` и `StatusStrip`. К элементу управления `StatusStrip` мы вернемся в этой главе позднее. Если их рассматривать в чистом виде, то `ToolStrip` и `MenuStrip` фактически представляют собой один и тот же элемент управления, поскольку `MenuStrip` унаследован непосредственно от элемента управления `ToolStrip`. Это озна-

чает, что MenuStrip умеет выполнять все, что может ToolStrip. Понятно, это означает также, что два эти элемента управления действительно хорошо сочетаются друг с другом.

Использование элемента управления MenuStrip

Кроме MenuStrip для заполнения меню можно использоваться еще несколько элементов управления. Наиболее часто применяются три элемента — ToolStripMenuItem, ToolStripDropDown и ToolStripSeparator, которые представляют конкретный способ просмотра элемента в меню или панели инструментов. ToolStripMenuItem представляет одиночную запись в меню, ToolStripDropDown — элемент, который при щелчке на нем отображает список других элементов, а ToolStripSeparator — горизонтальную или вертикальную разделительную линию в меню или панели инструментов.

Существует также еще один вид меню, который будет кратко рассмотрен после MenuStrip — ContextMenuStrip. Он представляет контекстное меню, которое открывается в результате щелчка на элементе правой кнопкой мыши и, как правило, отображает информацию, связанную с этим элементом.

Давайте приступим к следующему практическому занятию и создадим первый пример в этой главе.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Профессиональные меню за пять секунд

Первый пример в значительной степени является демонстрационным и призван просто ознакомить с особенностями элементов управления, которые действительно прекрасно подходят для создания стандартных меню, выглядящих и ведущих себя должным образом.

1. Создайте новое Windows-приложение Professional Menus в каталоге C:\BegVCSharp\Chapter16.
2. Перетащите экземпляр элемента управления MenuStrip из панели инструментов на поверхность проектирования.
3. Щелкните на треугольнике у правого края MenuStrip в верхней части диалогового окна, чтобы открыть окно действий.
4. Щелкните на маленьком треугольнике в верхнем правом углу меню, а затем на ссылке Insert Standard Items (Вставить стандартные элементы).

Вот и все. Если развернуть меню File (Файл), оно будет содержать привычные элементы, включая клавиши быстрого доступа и значки. Но пока оно не подкреплено никакой функциональностью — ее еще предстоит добавить. Меню можно изменять любым нужным образом. Это описано в последующих разделах.

Создание меню вручную

При перетаскивании элемента управления MenuStrip из панели инструментов на поверхность проектирования он помещается как в саму форму, так и в лоток элементов управления. Но инструмент управления можно редактировать непосредственно в форме. Чтобы создать новые элементы меню, достаточно поместить указатель в поле, помеченное надписью Type Here (Введите текст здесь).

При вводе заголовка меню в выделенном поле перед буквой, которая должна служить клавишей быстрого доступа к данному элементу меню, можно вставить знак амперсанда (&) — этот символ будет отображаться подчеркнутым, а для выбора пункта нужно будет нажать сочетание <Alt> и клавиши указанного символа.

В одном меню вполне возможно создать несколько элементов с одним и тем символом быстрого доступа. Правило гласит, что данный символ может использоваться для этой

цели только единожды в каждом раскрывающемся меню (например, один раз в меню File, один раз в меню View и т.д.). Если один и тот же символ быстрого доступа случайно назначить нескольким элементам одного и того же раскрывающегося меню, только ближайший к верхней части элемента управления элемент меню будет реагировать на нажатие данного символа.

При выборе элемента меню элемент управления автоматически отображает соответствующие элементы ниже и справа от текущего элемента. Ввод заголовка в любом из этих элементов ведет к созданию нового элемента, связанного с начальным. Именно так создаются раскрывающиеся меню.

Для создания горизонтальных линий, которые делят меню на группы, вместо `ToolStripMenuItem` необходимо использовать элемент управления `ToolStripSeparator`, но в действительности другой элемент управления вставлять не нужно. Вместо этого достаточно ввести символ – в качестве единственного символа заголовка элемента меню и Visual Studio автоматически интерпретирует его как разделитель и изменит тип элемента управления.

В следующем практическом занятии мы создадим меню, не прибегая к помощи Visual Studio для генерации его элементов.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание меню с нуля

В этом примере будут созданы меню `File` (Файл) и `Help` (Справка) с нуля. Меню `Edit` (Правка) и `Tools` (Сервис) предлагается создать самостоятельно.

1. Создайте новый проект приложения для Windows, назовите его `Manual Menus` и сохраните его в каталоге `C:\BegVCSharp\Chapter16`.
2. Перетащите элемент управления `MenuStrip` из панели инструментов на поверхность проектирования.
3. Щелкните в текстовом поле Type Here элемента управления `MenuStrip`, введите `&File` и нажмите клавишу <Enter>.
4. В текстовых полях под элементом File введите следующий текст:

```
&New
&Open
-
&Save
Save &As
-
&Print
Print Preview
-
E&xit
```

   Обратите внимание на то, как Visual Studio автоматически преобразует символы дефисов в линию, разделяющую элементы меню.

5. Щелкните на текстовой области справа от Files и введите `&Help`.
6. В текстовых полях под элементом Help введите следующий текст:

```
Contents
Index
Search
-
About
```

7. Вернитесь к меню File и определите клавиши быстрого доступа для его элементов. Для этого выберите элемент, который нужно определить, и найдите свойство `ShortcutKeys` в панели свойств. Щелчок на стрелке развертывания открывает маленькое окно, в котором можно определить клавиатурную комбинацию, связанную с данным элементом меню. Поскольку это меню является стандартным, следует использовать стандартные сочетания клавиш, но при создании дополнительных элементов можно выбирать любые другие клавиатурные комбинации. Установите свойства `ShortcutKeys` в меню File, как указано в следующей таблице:

Имя элемента	Свойства и значения
`&New`	`Ctrl+N`
`&Open`	`Ctrl+O`
`&Save`	`Ctrl+S`
`&Print`	`Ctrl+P`

8. Теперь в качестве завершающего штриха определите изображения. В меню File выберите пункт New (Создать) и щелкните на многоточии (`...`) слева от свойства `Image` в окне Properties, чтобы открыть диалоговое окно Select Resource (Выберите ресурс).

   Несомненно, наиболее трудоемкая часть создания этих меню связана с получением изображений, которые нужно отображать. В данном случае изображения входят в состав кода примеров для этой главы, но, как правило, их приходится рисовать самостоятельно или получать каким-либо способом.

9. Поскольку в настоящее время проект не содержит никаких ресурсов, список Entry (Элемент) пуст. Поэтому щелкните на кнопке Import (Импорт). Изображения для этого примера можно найти в в каталоге `Chapter16\Manual Menus\Images` исходного кода. Выберите все хранящиеся там файлы и щелкните на кнопке Open (Открыть). В данный момент мы выполняем редактирование элемента `New`, поэтому в списке Entry выберите запись `New image` (Новое изображение) и щелкните на кнопке OK.

10. Повторите шаг 9 для изображений кнопок Open (Открыть), Save (Сохранить), Save As (Сохранить как), Print (Печать) и Print Preview (Предварительный просмотр печати).

11. Запустите проект. Меню File можно выбирать, щелкая на нем либо нажимая комбинацию клавиш <Alt+F>, а доступ к меню Help осуществляется с помощью клавиатурной комбинации <Alt+H>.

Свойства элемента управления `ToolStripMenuItem`

Создавая меню, следует знать о нескольких дополнительных свойствах элемента управления `ToolStripMenuItem`, которые описаны в табл. 16.1. Информация, приведенная в табл. 16.1, не является исчерпывающей. Если хотите ознакомиться со всеми свойствами этого класса, обратитесь к документации .NET Framework SDK.

Таблица 16.1. Дополнительные свойства элемента управления `ToolStripMenuItem`

Свойство	Описание
`Checked`	Указывает, выбрано ли меню
`CheckOnClick`	Когда значение этого свойства равно `true`, метка в виде флажка либо добавляется, либо удаляется из позиции, расположенной слева от текста в элементе, которую в противном случае занимает изображение. Для определения состояния элемента меню следует использовать свойство `Checked`

Окончание табл. 16.1

Свойство	Описание
`Enabled`	Элемент, значение свойства `Enabled` которого установлено в `false`, будет затемнен и не может быть выбран
`DropDownItems`	Возвращает коллекцию элементов, которая используется в качестве раскрывающегося меню, связанного с данным элементом меню

Добавление функциональных возможностей меню

Теперь можно создавать меню, которые выглядят столь же привлекательно, как и доступные в Visual Studio. Остается только обеспечить, чтобы щелчок на них приводил к выполнению какого-то полезного действия. Понятно, что связанные со щелчком действия полностью зависят от программиста, но в следующем практическом занятии мы создадим очень простое приложение на основе предыдущего примера.

Чтобы приложение реагировало на выбор, выполненный пользователем, необходимо реализовать обработчики для одного из двух событий, отправляемых элементом управления `ToolStripMenuItems` (табл. 16.2).

Таблица 16.2. События элемента управления `ToolStripMenuItems`

Событие	Описание
`Click`	Отправляется каждый раз, когда пользователь щелкает на элементе. В большинстве случаев это событие, на которое требуется реагировать
`CheckedChanged`	Отправляется в результате щелчка на элементе, имеющем свойство `CheckOnClick`

Нам предстоит расширить предыдущий практический пример, добавив в диалоговое окно текстовое поле и реализовав несколько обработчиков событий. Мы добавим также еще одно меню — Format (Формат) — между меню Files и Help. В каталоге с кодом для данной главы этот проект называется `Manual Menus 2`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Обработка событий меню

1. Продолжите использование формы, созданной в предыдущем практическом занятии, и перетащите элемент управления `RichTextBox` на поверхность проектирования, а затем измените его имя на `richTextBoxText`. Установите `Fill` в качестве значения его свойства `Dock`.
2. Выберите `MenuStrip`, а затем в текстовой области введите `Format` после элемента меню `Help` и нажмите клавишу <Enter>.
3. Выберите элемент меню Format и перетащите его в позицию между Files и Help.
4. Добавьте в меню Format элемент меню с текстом Show Help Menu (Показывать меню Справка).
5. Установите значение свойства `CheckOnClick` элемента меню Show Help Menu в `true`. Установите значение его свойства `Checked` в `true`.
6. Выберите элемент меню `MenuItemShowHelpMenu` и добавьте обработчик события `CheckedChanged`, дважды щелкнув на событии в списке Events окна Properties.

7. Добавьте в обработчик события следующий код:

```
private void showHelpMenuToolStripMenuItem_CheckedChanged(object sender,
   EventArgs e)
   {
      ToolStripMenuItem item = (ToolStripMenuItem)sender;
      helpToolStripMenuItem.Visible = item.Checked;
   }
```

8. Дважды щелкните на элементах меню newToolStripMenuItem, saveToolStrip MenuItem и openToolStripMenuItem. Двойной щелчок на элементе ToolStripMenuItem в представлении конструктора ведет к добавлению события Click. Введите следующий код:

```
private void newToolStripMenuItem_Click(object sender, EventArgs e)
   {
      richTextBoxText.Text = "";
   }

   private void openToolStripMenuItem_Click(object sender, EventArgs e)
   {
      try
      {
         richTextBoxText.LoadFile(@"Example.rtf");
      }
      catch { }
   }

   private void saveToolStripMenuItem_Click(object sender, EventArgs e)
   {
      try
      {
         richTextBoxText.SaveFile("Example.rtf");
      }
      catch { }
   }
```

9. Запустите приложение. Щелчок на элементе меню Show Help Menu будет вызывать сокрытие или отображение меню Help в зависимости от состояния свойства Checked, и вы должны иметь возможность открывать, сохранять и очищать текст в текстовом поле.

Описание работы

Вначале осуществляется обработка события showHelpMenuToolStripMenuItem_ CheckedChanged. Обработчик этого события устанавливает значение свойства Visible элемента управления MenuItemHelp в true, если значение свойства Checked равно true. В противном случае MenuItemHelp получит значение false. В результате этот элемент управления работает в качестве переключателя видимости меню Help.

И, наконец, три обработчика событий Click соответственно очищают текст в элементе RichTextBox, сохраняют текст этого элемента в заранее определенном файле и открывают названный файл. Обратите внимание, что события Click и CheckedChanged идентичны в том, что они оба обрабатывают события, которые происходят при щелчке на элементе меню, но поведение конкретных элементов меню существенно различается и должно обрабатываться в зависимости от назначения элемента.

Панели инструментов

В то время как меню прекрасно подходят для предоставления доступа ко всему множеству функциональных возможностей приложения, некоторые элементы целесообразно помещать не только в меню, но и в панель инструментов. Панель инструментов позволяет с помощью одного щелчка получать доступ к таким часто используемым функциям, как Open (Открыть), Save (Сохранить) и т.п.

На рис. 16.1 показан набор панелей инструментов редактора Wordpad.

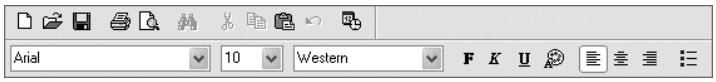


Рис. 16.1. *Панели инструментов Wordpad*

Обычно кнопка в панели инструментов отображается в виде рисунка без какого-либо текста, хотя могут существовать кнопки, для которых отображается и то, и другое. Примерами панелей инструментов без текста могут служить панели инструментов Wordpad (см. рис. 16.1), а примеры панелей инструментов, включающие текст, можно найти в браузере Internet Explorer. Изредка могут встречаться панели инструментов, которые кроме кнопок содержат также поля со списками и текстовые поля. Часто при помещении указателя мыши над кнопкой в панели инструментов, особенно, когда панель содержит только значки, отображается подсказка с информацией о назначении данной кнопки.

Элемент управления `ToolStrip`, как и `MenuStrip`, имеет профессиональный внешний вид. Когда пользователи видят панель инструментов, они ожидают, что ее можно перемещать в любую удобную позицию. `ToolStrip` дает возможность пользователям делать это — если вы разрешите.

После первоначального помещения элемента управления `ToolStrip` на поверхность проектирования формы он выглядит подобно рассмотренному ранее элементу управления `MenuStrip`, за исключением двух отличий: в крайней левой позиции элемента управления расположены четыре вертикальные точки, подобные присутствующим в меню Visual Studio. Эти точки показывают, что панель инструментов можно перемещать по окну родительского приложения и пристыковывать к его краям. Второе отличие в том, что по умолчанию панель инструментов отображает значки, а не текст, поэтому по умолчанию все ее элементы являются кнопками. Панель инструментов отображает раскрывающееся меню, которое позволяет выбирать тип элемента.

Одна из особенностей, полностью подобная `MenuStrip`, состоит в том, что окно Actions содержит ссылку Insert Standard Items (Вставить стандартные элементы). Щелчок на ней позволяет вставить не совсем те же элементы, которые можно было вставить с помощью `MenuStrip`, но мы получаем в свое распоряжение кнопки New (Создать), Open (Открыть), Save (Сохранить), Print (Печать), Cut (Вырезать), Copy (Копировать), Paste (Вставить) и Help (Справка). Вместо того чтобы полностью выполнять практическое упражнение, как было сделано ранее, рассмотрим некоторые свойства самого элемента управления `ToolStrip` и используемых в нем элементов управления.

Свойства элемента управления `ToolStrip`

Свойства элемента управления `ToolStrip` позволяют настроить способ и место его отображения. Помните, что в действительности этот элемент управления является базовым для рассмотренного ранее элемента управления `MenuStrip`, поэтому многие их свойства являются общими. Как и ранее, в табл. 16.3 описаны лишь некоторые свойства, представляющие особый интерес — чтобы ознакомиться с полным перечнем, обращайтесь к документации .NET Framework SDK.

Таблица 16.3. Часто используемые свойства элемента управления `ToolStrip`

Свойство	Описание
`GripStyle`	Управляет тем, должны ли четыре вертикальные точки отображаться в крайней левой позиции панели инструментов. Сокрытие этого манипулятора ведет к тому, что пользователи утрачивают возможность перемещения панели инструментов
`LayoutStyle`	Управляет способом отображения элементов в панели инструментов. По умолчанию они отображаются горизонтально
`Items`	Содержит коллекцию всех элементов панели инструментов
`ShowItemToolTip`	Определяет, нужно ли отображать подсказки для элементов панели инструментов
`Stretch`	По умолчанию панель инструментов лишь немного шире или выше содержащихся в ней элементов. Если значение свойства `Stretch` установлено в `true`, панель инструментов будет заполнять всю длину своего контейнера

Элементы элемента управления `ToolStrip`

В `ToolStrip` можно использовать множество элементов управления. Ранее уже отмечалось, что панель инструментов должна иметь возможность содержать кнопки, поля со списками и текстовые поля. Как легко догадаться, существуют элементы управления для каждого из этих элементов, однако имеется также несколько других элементов, которые описаны в табл. 16.4.

Таблица 16.4. Элементы управления, используемые в `ToolStrip`

Элемент управления	Описание
`ToolStripButton`	Представляет кнопку. Это свойство можно использовать для кнопок с текстом или без него
`ToolStripLabel`	Представляет метку. Этот элемент управления может также отображать изображения — т.е., его можно использовать для отображения статического изображения перед другим элементом управления, который не выводит информацию о себе, таким как текстовое поле или поле со списком
`ToolStripSplitButton`	Отображает кнопку с расположенной слева от нее кнопкой развертывания, щелчок на которой вызывает отображение меню под ней. Меню не разворачивается при щелчке на кнопочной части элемента управления
`ToolStripDropDownButton`	Аналогичен элементу управления `ToolStripSplitButton`. Единственное различие в том, что кнопка развертывания заменена изображением направленной вниз стрелки. Меню разворачивается при щелчке на любой части элемента управления
`ToolStripComboBox`	Отображает поле со списком
`ToolStripProgressBar`	Вставляет индикатор хода работ в панель инструментов
`ToolStripTextBox`	Отображает текстовое поле
`ToolStripSeparator`	Создает горизонтальные или вертикальные разделители элементов. Этот элемент управления уже рассматривался ранее

В следующем практическом занятии мы расширим пример использования меню, добавив в него панель инструментов. Панель инструментов будет содержать стандартные элементы и три дополнительных кнопки: Bold (Полужирный), Italic (Курсив) и Underline (Подчеркнутый). Она будет содержать также поле со списком для выбора шрифта. (Изображения, используемые в этом примере для кнопки выбора шрифта, можно найти в коде примеров для этой главы.)

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Расширение панели инструментов

Чтобы дополнить предыдущий пример панелями инструментов, выполните следующие действия.

1. Продолжите работу с примером, созданным в предыдущем практическом занятии, и удалите элемент `ToolStripMenuItem`, который был использован в меню Format. Выберите опцию Show Help Menu и нажмите клавишу <Delete>. Добавьте вместо него следующие три элемента `ToolStripMenuItems` и измените значение свойства `CheckOnClick` каждого из них на `true`:
   - `Bold`
   - `Italic`
   - `Underline`
2. Добавьте на форму элемент управления `ToolStrip`. В окне Actions Window (Окно действий) щелкните на опции Insert Standard Items (Вставить стандартные элементы). Выберите и удалите элементы Cut, Copy, Paste и следующий за ними элемент разделителя Separator. При вставке элемента управления `ToolStrip` стыковка элемента управления `RichTextBox` может перестать выполняться правильно. В этом случае измените стиль свойства `Dock` на `none` и вручную измените размер элемента управления, чтобы он заполнял форму. Затем измените значение свойства `Anchor` на `Top, Bottom, Left, Right`.
3. Создайте три новых кнопки и разделитель в конце панели инструментов, три раза выбрав опцию Button и один раз опцию Separator. (Чтобы добраться до этих опций, щелкните на последнем элементе в `ToolStrip`.)
4. Создайте два последних элемента, выбрав опцию `ComboBox` в раскрывающемся списке, а затем добавив разделитель в качестве последнего элемента.
5. Выберите элемент Help и перетащите его из текущей позиции в позицию последнего элемента в панели инструментов.
6. Три первых кнопки станут соответственно кнопками Bold, Italic и Underline. Назначьте им имена, как показано в следующей таблице:

ToolStripButton	**Имя**
Кнопка Bold	`boldToolStripButton`
Кнопка Italic	`italicToolStripButton`
Кнопка Underline	`underlineToolStripButton`
Поле со списком	`fontsToolStripComboBox`

7. Выберите кнопку Bold, щелкните на многоточии (`...`) в свойстве `Image`, выберите переключатель Project Resource File (Файл ресурса проекта) и щелкните на кнопке Import (Импортировать). Если вы пользуетесь исходным кодом примеров, исполь-

зуйте три изображения из каталога Chapter16\Toolbars\Images: BLD.ico, ITL.ico и UNDRLN.ico. Обратите внимание, что .ico не входит в число расширений, поддерживаемых по умолчанию Visual Studio. Поэтому при поиске значков в раскрывающемся списке необходимо выбрать опцию Show All Files (Показать все файлы).

8. Выберите BLD.ico в качестве изображения для кнопки Bold.
9. Выберите кнопку Italic и измените ее изображение на ITL.ico.
10. Выберите кнопку Underline и измените ее изображение на UNDRLN.ico.
11. Выберите элемент ToolStripComboBox. В окне Properties установите свойства, как показано в следующей таблице:

Свойство	Значение
Items	MS Sans Serif Times New Roman
DropDownStyle	DropDownList

12. Установите значение свойства CheckOnClick каждой из кнопок Bold, Italic и Underline в true.
13. Чтобы выбрать начальный элемент в поле со списком, добавьте следующий код в конструктор класса:

```
public Form1()
{
   InitializeComponent();

   fontsToolStripComboBox.SelectedIndex = 0;
}
```

14. Для запуска примера нажмите клавишу <F5>. Должно открыться диалоговое окно, подобное показанному на рис. 16.2.

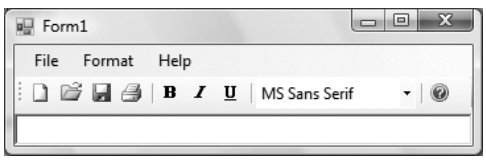


Рис. 16.2. *Приложение с панелью инструментов*

Добавление обработчиков событий

У нас уже имеются обработчики событий для элементов меню Save, New и Open, а кнопки панели инструментов должны себя вести совершенно так же, как элементы меню. Этого легко добиться за счет назначения событиям Click кнопок панели инструментов обработчиков, которые уже используются элементами меню. Определите события, как показано в следующей таблице:

ToolStripButton	Событие
New	newToolStripMenuItem_Click
Open	openToolStripMenuItem_Click
Save	saveToolStripMenuItem_Click

Теперь пора добавить обработчики для кнопок Bold, Italic и Underline. Поскольку эти кнопки являются кнопками флажков, вместо события `Click` нужно использовать событие `CheckedChanged`, поэтому добавьте это событие для каждой из трех названных кнопок. Добавьте следующий код:

```
private void boldToolStripButton_CheckedChanged(object sender, EventArgs e)
{
   Font oldFont, newFont;
   bool checkState = ((ToolStripButton)sender).Checked;
   oldFont = this.richTextBoxText.SelectionFont;
   if (!checkState)
     newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
   else
     newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);
   richTextBoxText.SelectionFont = newFont;
   richTextBoxText.Focus();
   boldToolStripMenuItem.CheckedChanged -= new
      EventHandler(boldToolStripMenuItem_CheckedChanged);
   boldToolStripMenuItem.Checked = checkState;
   boldToolStripMenuItem.CheckedChanged += new
      EventHandler(boldToolStripMenuItem_CheckedChanged);
}
private void italicToolStripButton_CheckedChanged(object sender, EventArgs e)
{
   Font oldFont, newFont;
   bool checkState = ((ToolStripButton)sender).Checked;
   oldFont = this.richTextBoxText.SelectionFont;
   if (!checkState)
     newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
   else
     newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);
   richTextBoxText.SelectionFont = newFont;
   richTextBoxText.Focus();
   italicToolStripMenuItem.CheckedChanged -= new
      EventHandler(italicToolStripMenuItem_CheckedChanged);
   italicToolStripMenuItem.Checked = checkState;
   italicToolStripMenuItem.CheckedChanged += new
      EventHandler(italicToolStripMenuItem_CheckedChanged);
}
private void UnderlineToolStripButton_CheckedChanged(object sender, EventArgs e)
{
   Font oldFont, newFont;
   bool checkState = ((ToolStripButton)sender).Checked;
   oldFont = this.richTextBoxText.SelectionFont;
   if (!checkState)
     newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
   else
     newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);
   richTextBoxText.SelectionFont = newFont;
   richTextBoxText.Focus();
   underlineToolStripMenuItem.CheckedChanged -= new
      EventHandler(underlineToolStripMenuItem_CheckedChanged);
   underlineToolStripMenuItem.Checked = checkState;
   underlineToolStripMenuItem.CheckedChanged += new
      EventHandler(underlineToolStripMenuItem_CheckedChanged);
}
```

Фрагмент кода ***Chapter16\Toolbars\Form1.cs***

Обработчики событий просто устанавливают нужный стиль для шрифта, используемого в элементе управления RichTextBox. Три последние строки каждого из трех методов выполняют действия с соответствующим элементом меню. Первая строка удаляет обработчик события из элемента меню. Это предотвращает запуск каких-либо событий при переходе к следующей строке. В результате состояние свойства Checked устанавливается в то же значение, что и у кнопки панели инструментов. И, наконец, выполняется переустановка обработчика события.

Обработчики событий элементов меню должны просто устанавливать свойство Checked для кнопок панели инструментов, предоставляя выполнение всех остальных необходимых действий обработчикам кнопок панели инструментов. Добавьте обработчики события CheckedChanged и введите следующий код:

```
private void boldToolStripMenuItem_CheckedChanged(object sender, EventArgs e)
{
   boldToolStripButton.Checked = boldToolStripMenuItem.Checked;
}

private void italicToolStripMenuItem_CheckedChanged(object sender, EventArgs e)
{
   italicToolStripButton.Checked = italicToolStripMenuItem.Checked;
}

private void underlineToolStripMenuItem_CheckedChanged(object sender, EventArgs e)
{
   underlineToolStripButton.Checked = underlineToolStripMenuItem.Checked;
}
```

Теперь осталось только предоставить пользователям возможность выбора семейства шрифтов из элемента управления ComboBox. Каждый раз, когда пользователь изменяет выбор в элементе управления ComboBox, приложение генерирует событие SelectedIndexChanged, поэтому добавьте обработчик для этого события:

```
private void fontsToolStripComboBox_SelectedIndexChanged(object sender,
  EventArgs e)
{
   string text = ((ToolStripComboBox)sender).SelectedItem.ToString();
   Font newFont = null;
   if (richTextBoxText.SelectionFont == null)
      newFont = new Font(text, richTextBoxText.Font.Size);
   else
      newFont = new Font(text, richTextBoxText.SelectionFont.Size,
                         richTextBoxText.SelectionFont.Style);
   richTextBoxText.SelectionFont = newFont;
}
```

Запустите приложение. Появилась возможность определять для текста полужирный, курсивный и подчеркнутый шрифт в панели инструментов. Обратите внимание, что при выборе или отмене выбора кнопки в панели инструментов в меню происходит установка или снятие флажка соответствующего элемента.

Элемент управления StatusStrip

Последним в небольшом семействе строковых элементов управления является StatusStrip. Он представляет строку, отображаемую в нижней части диалоговых окон многих приложений. Как правило, эта строка служит для отображения краткой информации о текущем состоянии приложения. Наглядным примером может быть отображение в строке состояния приложения Word текущей страницы, столбца, строки и т.п. во время ввода текста.

Элемент управления StatusStrip является производным от ToolStrip. Поэтому результат перетаскивания этого элемента управления поверх формы должен выглядеть достаточно знакомо. Три из четырех элементов управления, которые могут быть использованы в элементе управления StatusStrip — ToolStripDropDownButton, ToolStripProgressBar и ToolStripSplitButton — уже были представлены ранее. Поэтому остается рассмотреть только один элемент управления, характерный для StatusStrip: StatusStripStatusLabel, который является также используемым по умолчанию.

Свойства элемента управления StatusStripStatusLabel

Элемент StatusStripStatusLabel служит для предоставления пользователю краткой информации о текущем состоянии приложения в виде текста и изображений. Поскольку в действительности метка — очень простой элемент управления, количество его свойств, которые мы рассмотрим, не слишком велико. Два свойства, описанные в табл. 16.5, не являются специфичными для метки, но они могут и должны применяться с определенной пользой.

Таблица 16.5. Свойства StatusStripStatusLabel

Свойство	Значение
AutoSize	Свойство AutoSize включено по умолчанию, что в действительности не слишком естественно, поскольку нежелательно, чтобы метки в строке состояния перемещались по ней только из-за изменения текста в одной из них. Если только информация, отображаемая на метке, не является статичной, значение этого свойства всегда следует изменять на false
DoubleClickEnable	Это свойство указывает, будет ли генерироваться событие DoubleClick, что означает предоставление пользователям еще одного места для изменения чего-либо в приложении. Пример такого подхода — предоставление пользователям возможности двойного щелчка на панели, содержащей слово “Bold” для включения или отключения выделения текста полужирным.

В следующем практическом занятии мы создадим простую строку состояния для рассматриваемого примера. Строка состояния будет содержать четыре панели, три из которых отображают изображение и текст, а последняя — только текст.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с элементом управления StatusStrip

Выполните следующие действия, чтобы расширить возможности созданного нами простого текстового редактора.

1. Дважды щелкните на элементе StatusStrip в панели инструментов ToolBox, чтобы добавить его в диалоговое окно. Возможно, придется изменить размеры элемента RichTextBox на форме.
2. Щелкните на кнопке с символом многоточия (...) рядом со свойством Items элемента управления StatusStrip в окне Properties. В результате откроется редактор Items Collection Editor (Редактор коллекции элементов).
3. Щелкая на кнопке Add (Добавить) четыре раза, добавьте четыре панели в элемент управления StatusStrip. Установите для панелей свойства, как показано в следующей таблице.

Панель	Свойство	Значение
1	Name	toolStripStatusLabelText
	Text	Clear this property
	AutoSize	False
	DisplayStyle	Text
	Font	Arial; 8.25pt; style=Bold
	Size	259, 17
	TextAlign	Middle Left
2	Name	toolStripStatusLabelBold
	Text	Bold
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	47, 17
	Image	BLD
	ImageAlign	Middle-Center
3	Name	toolStripStatusLabelItalic
	Text	Italic
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	48, 17
	Image	ITL
	ImageAlign	Middle-Center
4	Name	toolStripStatusLabelUnderline
	Text	Underline
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	76, 17
	Image	UNDRLN
	ImageAlign	Middle-Center

4. Добавьте следующую строку в конец обработчика событий `boldToolStripButton_CheckedChanged`:

```
toolStripStatusLabelBold.Enabled = checkState;
```

5. Добавьте следующую строку в конец обработчика событий `italicToolStripButton_CheckedChanged`:

```
toolStripStatusLabelItalic.Enabled = checkState;
```

6. Добавьте следующую строку в конец обработчика событий `underlineToolStripButton_CheckedChanged`:

```
toolStripStatusLabelUnderline.Enabled = checkState;
```

7. Выберите элемент управления `RichTextBox` и добавьте в код событие `TextChanged`. Введите следующий код:

```
private void richTextBoxText_TextChanged(object sender, EventArgs e)
{
   toolStripStatusLabelText.Text = "Number of characters: " +
                                   richTextBoxText.Text.Length;
}
```

При запуске приложения должно открыться диалоговое окно, подобное показанному на рис. 16.3.

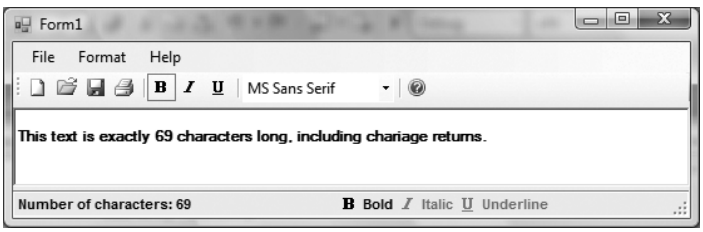


Рис. 16.3. *Приложение со строкой состояния*

Приложения SDI и MDI

Традиционно существуют три разновидности разрабатываемых Windows-приложений.

- **Приложения на основе диалоговых окон.** Эти приложения представляются пользователю как единственное диалоговое окно, в котором доступна вся функциональность.
- **Однодокументные интерфейсы (Single-document interfaces — SDI).** Эти приложения представляются пользователю в виде одного окна с меню и одной или несколькими панелями инструментов; в этом окне пользователь может выполнять ту или иную задачу.
- **Многодокументные интерфейсы (Multiple-document interfaces — MDI).** Эти приложения представляются пользователю так же, как SDI-приложения, но могут одновременно содержать несколько открытых окон.

Обычно приложения на основе диалоговых окон представляют собой небольшие, служащие единственной цели приложения, ориентированные на решение конкретной задачи, которая требует ввода минимума данных пользователем или же ориентирована на работу с очень специфичным типом данных. Примером такого приложения может служить калькулятор Windows.

Однодокументные интерфейсы обычно предназначены для решения одной конкретной задачи, поскольку они позволяют пользователям для работы загружать в приложение единственный документ. Обычно решение этой задачи связано с интенсивным взаимодействием с пользователем, и зачастую пользователям требуется возможность сохранения или

загрузки результатов своей работы. Показательными примерами SDI-приложений служат WordPad и Paint, поставляемые в составе ОС Windows. Простой текстовый редактор, созданный в этой главе — еще один пример SDI-приложения.

Однако в каждый конкретный момент времени эти приложения позволяют работать только с одним открытым документом. Поэтому, если пользователю требуется открыть второй документ, ему придется открыть новый экземпляр SDI-приложения, который никак не будет связан с первым экземпляром. Любые изменения в конфигурации одного экземпляра не переносятся в другой экземпляр. Например, в одном экземпляре Paint для рисования можно установить красный цвет, но при открытии второго экземпляра этого приложения цветом рисования будет заданный по умолчанию — черный.

Многодокументные интерфейсы во многом подобны SDI-приложениям, за исключением того, что в любой момент времени они могут содержать более одного открытого документа в различных окнах. Верным признаком MDI-приложения может служить наличие меню Window (Окно) непосредственно перед меню Help (Справка) в строке меню. Visual Studio является примером усовершенствованного MDI-приложения. В среде Visual Studio каждый сеанс конструирования и редактирования открывается в одном приложении, а меню и панели инструментов видоизменяются в соответствии с текущим выбором.

Построение MDI-приложений

Что же необходимо для создания многодокументного интерфейса? Во-первых, задача, которую должны быть в состоянии выполнять пользователи, должна быть такой, которая может требовать работы с несколькими открытыми документами. Показательный пример такого приложения — текстовый редактор или программа просмотра текстовых файлов. Во-вторых, нужно предоставить панели инструментов для выполнения наиболее часто выполняемых в приложении задач, таких как установка стиля шрифта, загрузка и сохранение документов. В-третьих, необходимо предусмотреть меню, включающее в себя пункт Window (Окно), которое позволит пользователям позиционировать открытые окна относительно друг друга (в виде плитки или каскадным образом) и будет представлять список всех открытых окон. Еще одна характерная особенность MDI-приложений состоит в том, что когда окно открыто и это окно содержит меню, это меню должно быть интегрировано в главное меню приложения.

MDI-приложение состоит, по меньшей мере, из двух различных окон. Первое окно, которое мы создадим, называется *контейнером MDI*. Окно, которое может отображаться внутри этого контейнера, называется *дочерним MDI*. В этой главе для контейнера MDI также применяется термин "главное окно", а для дочернего MDI — "дочернее окно".

В следующем практическом занятии рассматривается небольшой пример выполнения всех этих действий. Затем мы перейдем к решению более сложных задач.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание MDI-приложения

Создание MDI-приложения мы начнем с того же, что предпринимается для любого другого приложения — с создания приложения Windows Forms в среде Visual Studio.

1. Создайте новое Windows-приложение `MDIBasic` в каталоге `C:\BegVCSharp\Chapter16`.
2. Чтобы изменить тип главного окна приложения с формы на контейнер MDI, установите значение свойства `IsMdiContainer` формы в `true`. Цвет фона формы изменится, указывая, что теперь она представляет собой всего лишь фон, на который не следует помещать видимые элементы управления (хотя эти и возможно и даже целесообразно в некоторых ситуациях, таких как создание областей стыковки для окон).

Выберите форму и установите ее свойства, как показано в следующей таблице.

Свойство	Значение
Name	frmContainer
IsMdiContainer	True
Text	MDI Basic
WindowState	Maximized

3. Чтобы создать дочернее окно, добавьте в проект новую форму, указав Windows Form (Форма Windows) в диалоговом окне, которое открывается путем выбора пункта меню Project⇨Add New Item (Проект⇨Добавить новый элемент). Назначьте форме имя frmChild.

4. Новая форма становится дочерним окном при установке ссылки на главное окно в свойстве MdiParent дочернего окна. Это свойство нельзя установить с помощью окна Properties. Для этого нужно использовать код. Измените конструктор новой формы следующим образом:

```
public frmChild(frmContainer parent)
{
   InitializeComponent();

   MdiParent = parent;
}
```

5. Прежде чем MDI-приложение сможет отображаться в самом общем режиме, осталось выполнить всего два действия. Необходимо указать контейнеру MDI, какие окна следует отображать, а затем обеспечить их отображение. Просто создайте новый экземпляр формы, которую нужно отобразить, а затем вызовите для нее метод Show(). Конструктор формы, отображаемой в качестве дочерней, должен связываться с родительским контейнером. Этого можно достичь, устанавливая экземпляр контейнера MDI в качестве значения свойства MdiParent этого конструктора. Измените конструктор родительской формы MDI следующим образом:

```
public frmContainer()
{
   InitializeComponent();
   frmChild child = newfrmChild(this);
   child.Show();
}
```

Описание работы

Весь код, необходимый для отображения дочерней формы, сосредоточен в конструкторах формы. Вначале рассмотрим конструктор дочернего окна:

```
public frmChild(MdiBasic.frmContainer parent)
{
   InitializeComponent();

   // Установка контейнера в качестве родительского элемента управления формы.
   this.MdiParent = parent;
}
```

Чтобы связать дочернюю форму с контейнером MDI, необходимо обеспечить регистрацию дочерней формы с помощью контейнера. Это достигается установкой свойства MdiParent формы, как показано в предыдущем коде. Обратите внимание, что используемый конструктор включает в себя параметр parent.

Поскольку C# не предоставляет определенные по умолчанию конструкторы для класса, который определяет собственный конструктор, приведенный код не позволяет создать экземпляр формы, не связанной с контейнером MDI.

И, наконец, необходимо отобразить форму. Это осуществляется в конструкторе контейнера MDI:

```
public frmContainer()
{
   InitializeComponent();

   frmChild child = newfrmChild(this);

   child.Show();
}
```

Мы создаем новый экземпляр дочернего класса и передаем конструктору объект `this`, который представляет текущий экземпляр класса контейнера MDI. Затем мы вызываем метод `Show()` на новом экземпляре дочерней формы. Вот и все! Если нужно отобразить более одного дочернего окна, для каждого окна достаточно повторить две строки, выделенные в приведенном коде.

Рис. 16.4. *Простое приложение MDI*

Теперь запустите код. Интерфейс должен выглядеть подобно показанному на рис. 16.4 (только первоначально форма MDI Basic будет развернута на весь экран).

Созданный интерфейс пользователя не является чем-то выдающимся, но он явно служит хорошей отправной точкой. В следующем практическом упражнении мы создадим простой текстовый редактор на основе того, что уже достигли в этой главе с помощью меню, панелей инструментов и строк состояния.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание текстового редактора MDI

Вначале создадим базовый проект, а затем посмотрим, что и как происходит.

1. Снова обратитесь к рассмотренному ранее примеру использования строки состояния. Переименуйте форму на `frmEditor` и измените значение ее свойства `Text` на `Editor`.
2. Добавьте в проект новую форму `frmContainer.cs` и установите для нее свойства, перечисленные в следующей таблице:

Свойство	Значение
`Name`	`frmContainer`
`IsMdiContainer`	`True`
`Text`	`Simple Text Editor`
`WindowState`	`Maximized`

3. Откройте файл `Program.cs` и в методе `Main` измените строку, содержащую оператор `Run`, следующим образом:

```
Application.Run(new frmContainer());
```

4. Измените конструктор формы `frmEditor`, как показано ниже:

```
public frmEditor(frmContainer parent)
{
   InitializeComponent();

   this.ToolStripComboBoxFonts.SelectedIndex = 0;
   MdiParent = parent;
}
```

5. Замените значение свойства `MergeAction` элемента меню с текстом `&File` на `Replace`, а значение этого же свойства элемента с текстом `&Format` — на `MatchOnly`.

   Измените значение свойства `AllowMerge` панели инструментов на `False`.

6. Добавьте элемент управления `MenuStrip` на форму `frmContainer`. Добавьте в него единственный элемент с текстом `&File`.

7. Измените конструктор формы `frmContainer` следующим образом:

```
public frmContainer()
{
   InitializeComponent();

   frmEditor newForm = new frmEditor(this);
   newForm.Show();
}
```

   Запустите приложение. Результат должен выглядеть так, как показано на рис. 16.5.

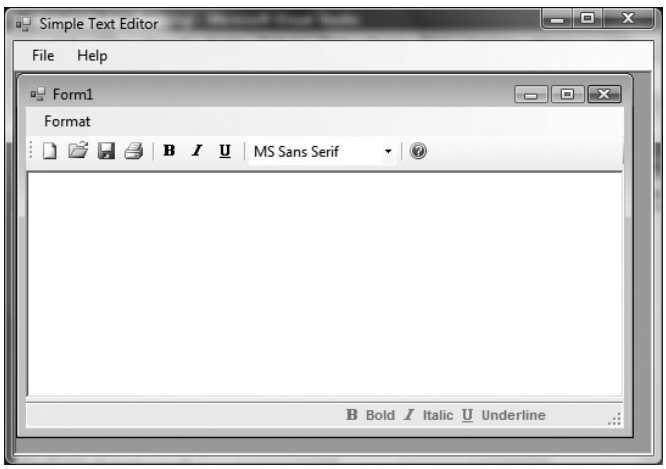


Рис. 16.5. *Приложение MDI с добавленным меню*

Описание работы

Обратите внимание на одну особенность. Создается впечатление, что меню File и Help были удалены из формы `frmEditor`. Выберите меню File в окне контейнера, и вы убедитесь, что теперь элементы меню диалогового окна `frmEditor` можно найти здесь.

Дочерние окна должны содержать те меню, которые специфичны для этих окон. Меню File должно быть общим для всех окон, а не присутствовать только в дочерних окнах. Причина этого становится понятной, если закрыть окно Editor — теперь меню File не содержит ни одного элемента! Меню File должно позволять вставлять элементы, которые специфичны для дочернего окна, когда это окно находится в фокусе, а все остальные элементы должны отображаться главным окном.

Свойства, перечисленные в табл. 16.6, управляют поведением элементов меню.

Таблица 16.6. Свойства, управляющие поведением элементов меню

Свойство	Описание
`MergeAction`	Определяет поведение элемента при его вставке в другое меню. Ниже перечислены возможные значения: `Append` — вызывает помещение элемента в последнюю позицию меню `Insert` — вставляет элемент непосредственно перед элементом, который соответствует критерию вставки; этим критерием может быть либо текст элемента, либо индекс `MatchOnly` — требуется соответствие, но элемент не будет вставлен `Remove` — удаляет элемент, который соответствует критерию вставки элемента `Replace` — элемент, соответствующий критерию, заменяется, а следующие за ним элементы размещаются вслед за вставляемым элементом
`MergeIndex`	Представляет позицию элемента меню относительно других элементов меню, с которым выполняется объединение. Если нужно управлять порядком объединяемых элементов, установите это значение равным `0`. В противном случае установите его в `-1`. При выполнении слияния это значение проверяется, и если оно не равно `-1`, то используется для сопоставления индексов элементов, а не их текста
`AllowMerge`	Установка значения этого свойства в `false` означает, что меню не будут объединяться

В следующем практическом занятии мы продолжим создание текстового редактора, изменяя способ объединения меню для отражения их принадлежности тому или иному окну.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Объединение меню

Чтобы изменить текстовый редактор, обеспечив использование меню и в контейнере, и в дочерних окнах, выполните перечисленные ниже действия.

1. Добавьте следующие четыре элемента меню в меню File формы `frmContainer`. Обратите внимание на скачкообразное изменение значений свойства `MergeIndex`.

Элемент	Свойство	Значение
`&New`	`MergeAction`	`MatchOnly`
	`MergeIndex`	`0`
	`ShortcutKeys`	`Ctrl + N`
`&Open`	`MergeAction`	`MatchOnly`
	`MergeIndex`	`1`
	`ShortcutKeys`	`Ctrl + O`
`-`	`MergeAction`	`MatchOnly`
`E&xit`	`MergeAction`	`MatchOnly`
	`MergeIndex`	`11`

2. Здесь нужен способ добавления новых окон, поэтому дважды щелкните на элементе меню New (Создать) и добавьте следующий код. Этот код совпадает с тем, что вводился в конструкторе для первого диалогового окна, предназначенного для отображения.

```
private void ToolStripMenuItemNew_Click(object sender, EventArgs e)
{
   frmEditor newForm = new frmEditor(this);
   newForm.Show();
}
```

3. В форме `frmEditor` удалите элемент Open из меню File. Модифицируйте свойства остальных элементов меню, как показано в следующей таблице.

Элемент	Свойство	Значение
&File	MergeAction	MatchOnly
	MergeIndex	-1
&New	MergeAction	MatchOnly
	MergeIndex	-1
-	MergeAction	Insert
	MergeIndex	2
&Save	MergeAction	Insert
	MergeIndex	3
Save &As	MergeAction	Insert
	MergeIndex	4
-	MergeAction	Insert
	MergeIndex	5
&Print	MergeAction	Insert
	MergeIndex	6
Print Preview	MergeAction	Insert
	MergeIndex	7
-	MergeAction	Insert
	MergeIndex	8
E&xit	Name	closeToolStripMenuItem
	Text	&Close
	MergeAction	Insert
	MergeIndex	9

4. Запустите приложение. Два меню File были объединены, но дочернее диалоговое окно все же сохранило меню File, содержащее один элемент — New.

Описание работы

Элементы, для которых значение свойства `MergeAction` установлено в `MatchOnly`, не перемещаются из одного меню в другое, но в случае элемента меню `&File` совпадение текста двух элементов ведет к объединению их элементов меню.

Элементы в меню File объединяются согласно значениям свойств `MergedIndex` соответствующих элементов. Значения свойства `MergeAction` тех из них, которые должны оставаться на месте, устанавливаются равными `MatchOnly`; значения остальных устанавливаются в `Insert`.

А что произойдет при щелчке на элементах меню New и Save (Сохранить) в двух различных меню? Вспомните, что меню New в дочернем диалоговом окне просто очищает текстовое поле, в то время как второе меню должно создавать новое диалоговое окно. Поскольку эти два меню должны принадлежать к различным окнам, не удивительно, что они оба работают, как предполагалось. Но как насчет элемента Save? Он перенесен из дочернего диалогового окна в родительское.

Откройте несколько диалоговых окон, введите в них какой-либо текст, а затем щелкните на меню Save. Откройте новое диалоговое окно и щелкните на меню Open (Открыть) (помните, что пункт Save всегда выполняет сохранение в том же файле). Выберите одно из других окон, щелкните на меню Save, вернитесь к новому диалоговому окну и снова щелкните на меню Open. Вы убедитесь, что элементы меню Save всегда переносятся в диалоговое окно, находящееся в фокусе. При каждом выборе диалогового окна меню снова объединяются.

Итак, мы добавили небольшой фрагмент кода в элемент New меню File диалогового окна `frmContainer` и удостоверились, что диалоговые окна были созданы. Одним из меню, которое присутствует едва ли не во всех MDI-приложениях, является меню Window (Окно). Оно позволяет упорядочивать диалоговые окна и часто отображает ту или иную форму их списка. В следующем практическом занятии мы добавим это меню в свой текстовый редактор.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Отслеживание окон

Чтобы расширить приложение, добавив в него возможность отображения всех открытых диалоговых окон и их упорядочения, выполните перечисленные ниже действия.

1. Добавьте в меню формы `frmContainer` новый элемент меню верхнего уровня `&Window`.
2. Добавьте в новое меню три элемента, описанные в следующей таблице.

Имя	Свойство *Text*
`tileToolStripMenuItem`	`&Tile`
`cascadeToolStripMenuItem`	`&Cascade`
`WindowsSeperatorMenuItem`	`-`

3. Выберите сам элемент `MenuStrip`, а не какой-то из отображаемых в нем элементов, и измените свойство `MDIWindowListItem` на `windowToolStripMenuItem`.
4. Дважды щелкните вначале на элементе `Tile`, а затем на элементе `Cascade`, чтобы добавить обработчики событий, и введите следующий код:

```
private void ToolStripMenuItemTile_Click(object sender, EventArgs e)
{
   LayoutMdi(MdiLayout.TileHorizontal);
}
private void ToolStripMenuItemCascasde_Click(object sender, EventArgs e)
{
   LayoutMdi(MdiLayout.Cascade);
}
```

5. Модифицируйте конструктор диалогового окна `frmEditor` следующим образом:

```
public frmEditor(frmContainer parent, int counter)
{
   InitializeComponent();
```

```
    this.ToolStripComboBoxFonts.SelectedIndex = 0;

    // Связывание с родительским окном.
    MdiParent = parent;
    Text = "Editor " + counter.ToString();
}
```

6. Добавьте приватную переменную-член в верхнюю часть кода формы `frmContainer` и измените конструктор и обработчик событий элемента меню New следующим образом:

```
public partial class frmContainer: Form
{
    private int counter;
    public frmContainer()
    {
        InitializeComponent();
        counter = 1;
        frmEditor newForm = new frmEditor(this, counter);
        newForm.Show();
    }
    private void ToolStripMenuItemNew_Click(object sender, EventArgs e)
    {
        frmEditor newForm = new frmEditor(this, ++counter);
        newForm.Show();
    }
```

Описание работы

Наиболее интересная часть этого примера связана с меню Window. Чтобы меню отображало список всех диалоговых окон, открытых в MDI-приложении, нужно всего лишь создать меню верхнего уровня и установить свойство `MdiWindowListItem`, чтобы оно указывало на это меню.

После этого .NET Framework будет добавлять в меню элемент для каждого отображенного диалогового окна. Элемент, который представляет текущее диалоговое окно, помечается флажком, и другое диалоговое окно можно выбрать, щелкая на нем в списке.

Остальные два элемента меню — Tile (Мозаика) and Cascade (Каскад) — демонстрируют метод, используемый формой: `MdiLayout`. Этот метод позволяет упорядочивать диалоговые окна стандартным образом.

Изменения в конструкторе и элементе New всего лишь обеспечивают нумерацию диалоговых окон. Запустите приложение, добавьте несколько окон и обратите внимание, что меню Window всегда отражает выбор того или иного окна.

Создание элементов управления

Иногда элементы управления, поставляемые с Visual Studio, просто не в состоянии удовлетворить все потребности. Причины этого могут быть самые различные — элементы прорисовываются не так, как желательно, ограничены в том или ином отношении, или же нужный элемент управления просто не существует. Осознавая это, Microsoft предоставляет средства для создания нужных элементов управления. В Visual Studio имеется проект Windows Control Library (Библиотека элементов управления Windows), который можно использовать для самостоятельного создания элемента управления.

Visual Studio позволяет разрабатывать два вида собственных элементов управления.

- **Пользовательские или составные элементы управления.** Новые элементы управления строятся на основе функциональности существующих элементов управления. В основном такие элементы управления создают для инкапсуляции функциональ-

ных возможностей в интерфейс пользователя элемента управления или для расширения интерфейса элемента управления за счет объединения нескольких элементов управления в один модуль.

- **Специальные элементы управления.** Эти элементы управления создаются, когда ни один из существующих не удовлетворяет конкретным нуждам — т.е. их создание начинается с нуля. Специальный элемент управления сам прорисовывает свой интерфейс пользователя, и при его создании никакие существующие элементы управления не применяются. Обычно потребность в создании такого элемента управления возникает, когда интерфейс пользователя нужного элемента управления не похож на интерфейс ни одного из доступных элементов управления.

В этой главе основное внимание уделено пользовательским элементам управления, поскольку разработка и рисование специального элемента управления выходят за рамки данной книги.

Элементы управления ActiveX, которые использовались в среде Visual Studio 6, существовали в специальном файле с расширением `.ocx`. По сути, эти файлы представляли собой COM-библиотеки DLL. В среде .NET элемент управления существует совершенно в том же виде, что и любая другая сборка, поэтому расширение `.ocx` исчезло, а элементы управления существуют в виде библиотек DLL.

Пользовательские элементы управления являются производными от класса `System.Windows.Forms.UserControl`. Этот базовый класс снабжает создаваемый элемент управления основной функциональностью, присущей элементу управления в среде .NET, оставляя программисту только задачу создания элемента управления. В качестве элемента управления могут выступать буквально любые компоненты — от меток причудливой формы до полнофункциональных таблиц. На рис. 16.6 новый элемент управления представлен нижним прямоугольником, `UserControl1`.

Пользовательские элементы управления являются производными от класса `System.Windows.Forms.UserControl`, а специальные элементы управления — от класса `System.Windows.Forms.Control`.

При работе от элементов управления ожидают определенного поведения. Если элемент управления не оправдывает описанные ниже ожидания, его использование вполне может разочаровать пользователя.

- Поведение элемента управления во время проектирования должно быть во много аналогично его поведению во время выполнения. Это означает, что если элемент управления состоит из элементов управления `Label` и `TextBox`, объединяемых для создания элемента управления `LabelTextbox`, и `Label`, и `TextBox` должны отображаться во время проектирования, а текст, введенный в элементе `Label`, должен также отображаться во время проектирования. Хотя в данном примере достаточно просто добиться выполнения этого условия, в более сложных случаях это может быть сопряжено с проблемами, когда придется искать подходящий компромисс.

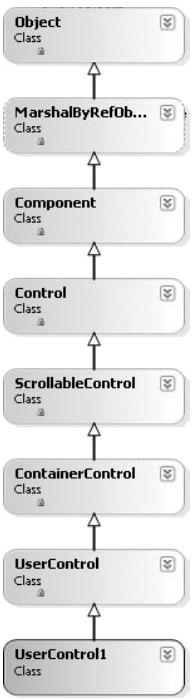


Рис. 16.6. *Иерархия классов для пользовательского элемента управления `UserControl1`*

- Доступ к свойствам элемента управления в визуальном конструкторе форм должен осуществляться логичным образом. Показательным в этом отношении примером может служить элемент управления `ImageList`, представляющий диалоговое окно, из которого пользователи могут выполнять обзор для выбора нужных изображений; как только изображения импортированы, они отображаются в диалоговом окне в списке.

Ниже рассматривается пример создания элементов управления. Будет создан элемент управления `LabelTextbox` и продемонстрированы основы создания проекта пользовательского элемента управления, создания свойств и событий, а также отладки элементов управления.

Как следует из имени элемента управления в последующем разделе, этот элемент управления объединяет в себе два существующих элемента, который за один проход выполняет чрезвычайно распространенную в программировании для Windows задачу: добавление на форму метки и текстового поля с позиционированием текстового поля относительно метки. Пользователь этого элемента управления будет ожидать от него описанное ниже поведение.

- Пользователю желательно иметь возможность размещать текстовое поле справа от метки или под ней. Если текстовое поле размещается справа от метки, пользователь должен иметь возможность указывать фиксированное расстояние между левым краем элементом управления и текстовым полем, чтобы текстовые поля можно было выравнивать, размещая их друг под другом.
- Пользователь должен иметь доступ к обычным свойствам и событиям текстового поля и метки.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Элемент управления `LabelTextbox`

Теперь, когда стоящие задачи ясны, запустите Visual Studio и создайте новый проект.

1. Создайте новый проект Windows Forms Control Library (Библиотека элементов управления Windows Forms) по имени `LabelTextbox` и сохраните его в каталоге `C:\BegVCSharp\Chapter16`.

При использовании версии Visual Studio Express данная опция может быть недоступной. В этом случае создайте новый проект Class Library (Библиотека классов) и добавьте пользовательский элемент управления в проект вручную через меню Project *(Проект).*

   Визуальный конструктор Forms Designer предоставляет поверхность проектирования, несколько отличающуюся от использованной ранее. Во-первых, она значительно меньше. Во-вторых, она вообще не выглядит как диалоговое окно. Пусть новый вид интерфейса не вводит вас в заблуждение — все работает, как обычно. Основное различие в том, что до сих пор мы помещали элементы управления в существующую форму, а теперь предстоит создать элемент управления, предназначенный для помещения в какую-либо форму.

2. Щелкните на поверхности проектирования и откройте свойства элемента управления. Измените значение свойства `name` элемента управления на `ctlLabelTextbox`.
3. Дважды щелкните на элементе управления `Label` в панели инструментов Toolbox, чтобы добавить его в создаваемый элемент управления, поместив в верхнем левом углу поверхности. Измените значение его свойства `Name` на `labelCaption`. Установите `Label` в качестве значения свойства `Text`.

4. Дважды щелкните на элементе TextBox в панели инструментов Toolbox, чтобы добавить его в создаваемый элемент управления. Измените значение его свойства Name на textBoxText.

Во время проектирования не известно, как пользователь пожелает разместить эти элементы управления, поэтому придется написать код, который будет позиционировать элементы управления Label и TextBox. Этот же код будет определять позицию элементов управления при помещении элемента управления LabelTextbox в форму.

Дизайн элемента управления выглядит не особенно удачно — мало того, что элемент управления TextBox заслоняет часть метки, так еще и поверхность всего составного элемента управления слишком велика. Однако это не имеет значения, поскольку в отличие от элементов, использованных до сих пор, то, что мы видим, не является тем, что мы получаем! Код, который мы собираемся добавить к элементу управления, будет изменять его внешний вид, но только при добавлении элемента управления на форму.

Пользователь должен иметь возможность выбора способа размещения элементов управления, и поэтому мы добавим к элементу управления не одно, а два свойства. Свойство Position позволяет пользователю выбирать одну из двух опций: Right (справа) и Below (внизу). Если пользователь выбирает опцию Right, в дело вступает еще одно свойство. Оно называется TextboxMargin, и его значение типа int представляет количество пикселей между левым краем элемента управления и позицией размещения элемента управления TextBox. Если пользователь указывает значение 0, при размещении правый край элемента управления TextBox выравнивается по правому краю элемента управления.

Добавление свойств

Чтобы предоставить пользователю возможность выбора между опциями Right и Below, начните с определения перечисления этих двух значений. Вернитесь к проекту элемента управления, перейдите в редактор кода и добавьте следующий код:

```
public partial class ctlLabelTextbox: UserControl
{

  public enum PositionEnum
  {
    Right,
    Below
  }
```

Это всего лишь обычное перечисление, описанное в главе 5. А теперь, что касается "магии": позиция должна быть свойством, которое пользователь сможет устанавливать как в коде, так и с помощью визуального конструктора. Мы достигнем этого за счет добавления свойства в класс ctlLabelTextbox. Однако вначале нужно создать два поля-члена, которые будут содержать значения, выбираемые пользователем:

```
private PositionEnum position = PositionEnum.Right;
private int textboxMargin = 0;
```

Затем добавьте свойство Position:

```
public PositionEnum Position
{
  get { return position; }
  set
  {
    position = value;
    MoveControls();
  }
}
```

Это свойство добавляется в класс подобно любому другому свойству. Если требуется вернуть свойство, мы возвращаем поле-член `position`; если же требуется изменить значение свойства `Position`, мы присваиваем значение `position` и вызываем метод `MoveControls()`. Несколько позже мы вернемся к рассмотрению метода `MoveControls()` — а пока достаточно знать, что этот метод позиционирует два элемента управления, анализируя значения `position` и `textboxMargin`.

Свойство `TextboxMargin` аналогично предыдущему, за исключением того, что оно работает с целочисленным значением:

```
public int TextboxMargin
{
   get { return textboxMargin; }
   set
   {
      textboxMargin = value;
      MoveControls();
   }
}
```

Добавление обработчиков событий

Прежде чем приступить к тестированию двух описанных свойств, нужно добавить также два обработчика событий. При помещении элемента управления на форму вызывается событие `Load`. Это событие применяется для инициализации элемента управления и любых ресурсов, которые он может использовать. В обработчике события `Load` производится перемещение элемента управления и изменение его размеров так, чтобы он аккуратно охватывал два содержащихся в нем элемента управления.

Второе добавляемое событие — `SizeChanged`. Оно вызывается при каждом изменении размеров элемента управления, и его обработка требуется для обеспечения правильности своей прорисовки самим элементом управления. Выберите элемент управления и добавьте два события: `SizeChanged` и `Load`.

Затем добавьте обработчики событий.

```
private void ctlLabelTextbox_Load(object sender, EventArgs e)
{
   labelCaption.Text = Name;
   Height = textBoxText.Height > labelCaption.Height ?
      textBoxText.Height : labelCaption.Height; MoveControls();
}

private void ctlLabelTextbox_SizeChanged(object sender, System.EventArgs e)
{
   MoveControls();
}
```

И снова мы вызываем метод `MoveControls()` для позиционирования элементов управления. Теперь, прежде чем снова проверять созданный элемент управления, необходимо рассмотреть этот метод:

```
private void MoveControls()
{
   switch (position)
   {
      case PositionEnum.Below:
         textBoxText.Top = labelCaption.Bottom;
         textBoxText.Left = labelCaption.Left;
         textBoxText.Width = Width;
         Height = textBoxText.Height + labelCaption.Height;
         break;
```

```
        case PositionEnum.Right:
          textBoxText.Top = labelCaption.Top;
          if (textboxMargin == 0)
          {
            int width = Width - labelCaption.Width - 3;
            textBoxText.Left = labelCaption.Right + 3;
            textBoxText.Width = width;
          }
          else
          {
            textBoxText.Left = textboxMargin + labelCaption.Width;
            textBoxText.Width = Width - textBoxText.Left;
          }
          Height = textBoxText.Height > labelCaption.Height ?
                                textBoxText.Height : labelCaption.Height;
          break;
      }
    }
```

Для определения того, должно ли текстовое поле размещаться под меткой или справа от нее, мы проверяем значение `position` в операторе `switch`. Если пользователь выбирает значение `Below`, верхний край текстового поля перемещается в позицию, расположенную под меткой. Затем левый край текстового поля перемещается к левому краю элемента управления, а его ширина устанавливается равной ширине элемента управления.

Если пользователь выбирает опцию `Right`, существуют две возможности. Если значение `TextboxMargin` равно `0`, вначале нужно определить ширину незанятой области элемента управления, которую можно предоставить текстовому полю. Затем левый край текстового поля прижимается к правому краю текста, а его ширина устанавливается такой, чтобы заполнить свободную область элемента управления. Если же пользователь указывает ширину границы, левый край текстового поля помещается в указанную позицию, а его ширина определяется соответствующим образом.

Теперь можно протестировать элемент управления. Но прежде чем двигаться дальше, постройте проект.

Отладка пользовательских элементов управления

Отладка пользовательского элемента управления значительно отличается от отладки Windows-приложения. Обычно достаточно добавить точку останова где-либо, нажать клавишу <F5> и посмотреть, что происходит. Если вы еще не знакомы с процессом отладки, обратитесь к главе 7.

Чтобы он мог себя отображать, элемент управления нуждается в контейнере, и нужно предоставить такой контейнер. Мы выполним это в следующем практическом занятии, создав проект Windows-приложения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Отладка пользовательских элементов управления**

1. В меню File выберите пункт Add⇨New Project (Добавить⇨Новый проект). В диалоговом окне Add New Project создайте новое Windows-приложение по имени `LabelTextboxTest`. Поскольку это приложение предназначено только для тестирования пользовательского элемента управления, имеет смысл создать проект внутри решения `LabelTextBox`.

   Теперь в окне Solution Explorer вы должны видеть два открытых проекта. Первый созданный проект, `LabelTextbox`, выделен полужирным. Это означает, что при попытке запуска решения отладчик будет пытаться использовать проект элемента управления в качестве стартового. Эта попытка окажется безуспешной, поскольку проект

элемента управления не является самостоятельным. Чтобы исправить эту ситуацию, щелкните правой кнопкой мыши на имени нового проекта — LabelTextboxTest — и в контекстном меню выберите пункт Set as StartUp Project (Установить в качестве стартового проекта). Если теперь запустить решение, проект Windows-приложения будет запущен без каких-либо ошибок.

2. Теперь верхняя часть панели инструментов Toolbox должна содержать вкладку LabelTextBox Components (Компоненты LabelTextBox). Visual Studio распознает, что решение содержит компоненты библиотеки Windows Control Library, и что, вполне вероятно, элементы управления, предоставляемые этой библиотекой, желательно использовать в других проектах. Дважды щелкните на элементе управления ctlLabelTextbox, чтобы добавить его к форме. Обратите внимание, что узел References (Ссылки) в окне Solution Explorer развернут. Это обусловлено тем, что Visual Studio только что автоматически добавила ссылку на проект LabelTextBox.
3. В коде поищите компонент ctlLabel. Поиск нужно выполнять во всем проекте. В результате вы натолкнетесь на "фоновый" файл Form.Designer.cs, в котором Visual Studio скрывает большую часть генерируемого ею кода. Помните, что этот файл никогда не следует редактировать непосредственно.
4. Поместите точку останова на следующую строку:

```
this.ctlLabelTextbox1 = new LabelTextbox.ctlLabelTextbox();
```

5. Запустите код. Как и следовало ожидать, выполнение кода останавливается в установленной точке останова. Теперь выполните вход в код (при использовании раскладок клавиатуры, заданных по умолчанию, нажмите для этого клавишу <F11>). При входе в код будет выполнен переход к конструктору нового элемента управления, что и требуется для отладки компонента. Здесь же можно помещать дополнительные точки останова. Чтобы запустить приложение, нажмите клавишу <F5>.

Расширение элемента управления LabelTextbox

Теперь, наконец, можно протестировать свойства элемента управления. Обратите внимание, что элементы управления внутри LabelTextbox перемещаются в соответствующие позиции при добавлении этого элемента управления на форму. Поскольку по умолчанию в качестве значения свойства Position установлено значение Right, текстовое поле размещается в элементе управления рядом с элементом Label. Если изменить значение свойства Position на Below, текстовое поле перемещается под метку.

Добавление дополнительных свойств

В данный момент элемент управления позволяет выполнить не слишком многое, поскольку, к сожалению, он лишен возможности изменения текста в метке и текстовом поле. Для устранения этого недостатка добавим два свойства: LabelText и TextboxText. Их добавление выполняется точно так же, как и двух предыдущих — откройте проект и добавьте следующий код:

```
public string LabelText
{
   get { return labelCaption.Text; }
   set
   {
      labelCaption.Text = labelText = value;
      MoveControls();
   }
}
```

```
public string TextboxText
{
   get { return textBoxText.Text; }
   set
   {
      textBoxText.Text = value;
   }
```

Нужно также объявить переменную-член `labelText` для хранения текста:

```
private string labelText = "";
public ctlLabelTextbox()
{
```

Чтобы вставить текст, достаточно присвоить текст свойству `Text` элементов управления `Label` и `TextBox` и вернуть значения свойств `Text`. При изменении текста метки необходимо вызывать метод `MoveControls()`, поскольку текст метки может влиять на позицию текстового поля. И напротив, текст, вставленный в текстовое поле, не вызывает перемещение элементов управления; если текст длиннее текстового поля, он исчезает из виду.

И, наконец, нужно следующим образом изменить событие `Load`:

```
private void ctlLabelTextbox_Load(object sender, EventArgs e)
{
   labelCaption.Text = labelText;
   Height = textBoxText.Height > labelCaption.Height ?
      textBoxText.Height : labelCaption.Height;
   MoveControls();
}
```

Событие `Load` устанавливает текст элемента управления `labelCaption` равным значению свойства. В результате текст, отображаемый во время разработки, отображается также и во время выполнения.

Добавление дополнительных обработчиков событий

Теперь пора подумать о том, какие события должен предоставлять элемент управления. Поскольку он является производным от класса `UserControl`, он наследует множество функциональных возможностей, не требующих специальной обработки. Однако существует ряд событий, таких как `KeyDown`, `KeyPress` и `KeyUp`, обработку которых нежелательно выполнять стандартным образом. Эти события необходимо изменить, поскольку пользователи будут ожидать их отправки при нажатии клавиши в текстовом поле. В существующем состоянии события отправляются, только если сам элемент управления находится в фокусе и пользователь нажимает клавишу.

Чтобы изменить это поведение, необходимо выполнить обработку событий, отправленных текстовым полем, и передать их пользователю. Добавьте события `KeyDown`, `KeyUp` и `KeyPress` для текстового поля и введите следующий код:

```
private void textBoxText_KeyDown(object sender, KeyEventArgs e)
{
   OnKeyDown(e);
}
private void textBoxText_KeyPress(object sender, KeyPressEventArgs e)
{
   OnKeyPress(e);
}
private void textBoxText_KeyUp(object sender, KeyEventArgs e)
{
   OnKeyUp(e);
}
```

Вызов метода `OnKeyXXX` влечет за собой вызов любых методов, подписанных на событие.

Добавление нестандартного обработчика события

При необходимости создания события, которое не существует ни в одном из базовых классов, придется проделать несколько больший объем работы. Создайте событие `PositionChanged`, которое будет происходить при изменении свойства `Position`. Для создания этого события требуется выполнение трех условий.

- Наличие подходящего делегата, который может быть использован для вызова методов, присваиваемых пользователем событию.
- Пользователь должен иметь возможность подписки на событие, посредством присваивания ему метода.
- Мы должны вызвать метод, который пользователь присвоил событию.

Здесь будет использоваться делегат `EventHandler`, предоставляемый .NET Framework. Как было описано в главе 13, он представляет собой специальный вид делегата, объявляемый собственным ключевым словом, `event`. Следующая строка объявляет событие и позволяет пользователю подписаться на него:

```
public event System.EventHandler PositionChanged;
public ctlLabelTextbox()
{
```

Теперь остается только сгенерировать событие. Поскольку оно должно происходить при изменении свойства `Position`, мы генерируем его в средстве доступа к свойству `Position`:

```
public PositionEnum Position
{
   get { return position; }
   set
   {
      position = value;
      MoveControls();
      if (PositionChanged != null)
         PositionChanged(this, new EventArgs());
   }
}
```

Вначале удостоверьтесь в наличии каких-либо подписчиков, проверив, не равно ли `PositionChanged` значению `null`. Если нет, следует вызвать методы.

Подписка на новое нестандартное событие осуществляется так же, как она выполнялась бы для любого другого, но при этом есть один небольшой нюанс: прежде чем событие будет отображено в окнах событий, необходимо построить элемент управления. После того как он построен, выберите элемент управления в форме в проекте `LabelTextboxTest` и дважды щелкните на событии `PositionChanged` в разделе Events (События) окна Properties. Затем добавьте следующий код в обработчик события:

```
private void ctlLabelTextbox1_PositionChanged(object sender, EventArgs e)
{
   MessageBox.Show("Changed");
}
```

В действительности нестандартный обработчик события не делает ничего особенного — он просто указывает на то, что позиция изменилась.

И, наконец, добавьте кнопку на форму, дважды щелкните на ней, чтобы добавить в проект обработчик ее события `Click`, а затем добавьте следующий код:

```
   private void buttonToggle_Click(object sender, EventArgs e)
   {
      ctlLabelTextbox1.Position = ctlLabelTextbox1.Position ==
         LabelTextbox.ctlLabelTextbox.PositionEnum.Right ?
```

```
            LabelTextbox.ctlLabelTextbox.PositionEnum.Below :
            LabelTextbox.ctlLabelTextbox.PositionEnum.Right;
    }
```

После запуска приложения позицию текстового поля можно будет изменять во время выполнения. При каждом перемещении текстового поля приложение вызывает событие `PositionChanged` и отображает `messagebox`.

На этом создание примера завершено. Приложение можно было бы немного усовершенствовать, но это оставляется для самостоятельной проработки.

Резюме

В этой главе мы начали с того, на чем остановились в предыдущей главе, и исследовали элементы управления `MainMenu` и `ToolBar`. Вы научились создавать приложения MDI и SDI и узнали, как меню и панели инструментов используются в этих приложениях. Затем мы перешли к созданию собственного элемента управления: разработке свойств, интерфейса пользователи и событий элемента управления. Следующая глава посвящена развертыванию Windows-приложений.

Упражнения

1. Используя пример `LabelTextbox` в качестве основы, создайте новое свойство `MaxLength`, хранящее максимальное количество символов, которые могут быть введены в текстовом поле. Затем создайте два новых события `MaxLengthChanged` и `MaxLengthReached`. Событие `MaxLengthChanged` должно генерироваться при изменении свойства `MaxLength`, а событие `MaxLengthReached` — когда в результате ввода символом пользователем длина текста в текстовом поле становится равной значению свойства `MaxLength`.
2. Элемент управления `StatusBar` включает свойство, которое позволяет пользователю дважды щелкнуть на поле в строке и тем самым запустить событие. Измените пример `StatusBar`, чтобы пользователь мог двойным щелчком на строке состояния выделять текст полужирным, курсивным и подчеркнутым шрифтом. Обеспечьте, чтобы текст **Bold**, отображаемый в панели инструментов, меню и строке состояния, всегда выделялся полужирным шрифтом, когда эта опция активизирована, и не выделялся в противном случае. Выполните это же для опций *Italic* (Курсив) и Underlined (Подчеркнутый).

   Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Меню	Используйте элемент управления `MenuStrip` для отображения профессионально выглядящих меню в формах.
Панели инструментов	Используйте элемент управления `ToolStrip` для отображения профессионально выглядящих панелей инструментов в формах.
Строки состояния	Элемент управления `StatusStrip` предоставляет средства для отображения информации о текущем состоянии приложения.
MDI-приложения	Приложения MDI позволяют дополнительно расширять возможности таких приложений, как текстовый редактор.
Пользовательские элементы управления	Пользовательские элементы управления создаются на основе существующих элементов управления.

17

Развертывание Windows-приложений

В ЭТОЙ ГЛАВЕ...

- Обзор вариантов развертывания
- Развертывание Windows-приложений с помощью технологии ClickOnce
- Создание пакета развертывания для программы установки Windows
- Установка приложения с помощью программы установки Windows

Существует несколько способов развертывания Windows-приложений. Простые приложения можно устанавливать с помощью базового развертывания посредством утилиты `xcopy`, но для установки сотен клиентских приложений этот вид развертывания не подходит. Для таких ситуаций предусмотрены две возможности: применение развертывания ClickOnce либо программы установки Microsoft Windows.

При развертывании ClickOnce установка приложения осуществляется щелчком на ссылке, ведущей на некоторый веб-сайт. В ситуациях, когда пользователь должен выбирать каталог для установки приложения или когда требуется изменение каких-либо записей системного реестра, следует применять развертывание с помощью программы установки Windows.

В этой главе освещены оба варианта установки Windows-приложений.

Обзор процесса развертывания

Развертывание — это процесс установки приложений в целевых системах. Традиционно это осуществлялось с помощью программы установки. Процесс установки ста или даже тысячи клиентских приложений может занимать очень много времени. Для облегчения этой задачи системный администратор может создать пакетные сценарии, автоматизирующие выполнение необходимых действий. Но и в этом случае приходится все же выполнять огромный объем работы по установке и поддержке различных клиентских ПК и различных версий операционной системы.

В связи с этими проблемами многие компании преобразуют свои внутрисетевые приложения в веб-приложения, несмотря на то, что Windows-приложения предлагают интерфейс пользователя с существенно большими возможностями. Веб-приложения нужно развернуть только на сервере, а клиент автоматически получает новейший интерфейс пользователя.

Создание приложений Silverlight — одна из возможностей веб-развертывания сложных клиентских приложений.

Применение установки ClickOnce позволяет предотвратить многие из этих проблем развертывания Windows-приложений. В этом случае приложения могут быть установлены посредством щелчка на ссылке внутри веб-страницы. Пользователь клиентской системы не нуждается в административных привилегиях, поскольку приложение устанавливается в каталоге, выделенном данному пользователю. Применение технологии ClickOnce позволяет устанавливать приложения со сложным пользовательским интерфейсом. Приложение устанавливается на компьютере клиента, поэтому по завершении установки необходимость сохранения соединения с клиентской системой отпадает. Иначе говоря, приложение может эксплуатироваться в автономном режиме. В результате значок приложения становится доступным в меню Start (Пуск), проблемы безопасности легче поддаются решению, а приложение может быть легко удалено.

Замечательная особенность технологии ClickOnce состоит в том, что обновления могут производиться автоматически при запуске клиентского приложения или в фоновом режиме во время его работы.

Однако развертывание ClickOnce сопряжено с рядом ограничений: эта технология не может применяться, если компоненты совместного использования требуется установить в глобальный кэш сборок, если приложение нуждается в COM-компонентах, которые требуют изменений в системном реестре, или если пользователи должны иметь возможность выбора каталога для установки приложения. В подобных случаях следует использовать программу установки Windows, предлагающую традиционный способ установки Windows-приложений. Однако прежде чем приступать к работе с пакетами программы установки Windows, в следующем разделе мы рассмотрим развертывание ClickOnce.

Развертывание ClickOnce

При использовании технологии развертывания ClickOnce запуск программы установки в клиентской системе не требуется. Пользователю клиентской системы достаточно щелкнуть на ссылке на веб-странице, и приложение будет автоматически установлено. По завершении установки приложения клиентская система может работать в автономном режиме — ей не требуется доступ к серверу, с которого приложение было установлено.

Установка ClickOnce может быть выполнена с веб-сайта, разделяемого ресурса в сети или места хранения файла (например, компакт-диска). При использовании технологии ClickOnce приложение устанавливается в клиентской системе, доступно через ярлыки быстрого доступа меню Start и может быть удалено из системы через диалоговое окно Add/Remove Programs (Установка и удаление программ).

Развертывание ClickOnce описывается *файлами манифестов*. Манифест приложения описывает приложение и требуемые им разрешения. Манифест развертывания содержит информацию о конфигурации развертывания, такую как политики обновления. В практических примерах этого раздела мы выполним конфигурирование ClickOnce-развертывания MDI-редактора, который был создан в главе 16, и снова обратимся к этим файлам кода.

Подготовка к развертыванию ClickOnce

В следующем практическом занятии мы изменим имя приложения и определим набор полезных настроек сборки.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Подготовка приложения

1. Откройте в Visual Studio пример MDI-редактора, созданный в главе 16. Если по какой-либо причине этот пример не был создан, скопируйте всю папку `MDI Editor` из кода примеров для данной главы. С помощью пункта меню File⇨Open⇨Project/Solution (Файл⇨открыть⇨Проект/Решение) в Visual Studio откройте файл решения `MDI Editor.sln`.
2. В окне Solution Explorer выберите элемент Properties (Свойства) для проекта, а затем перейдите на вкладку Application (Приложение), показанную на рис. 17.1.
3. Измените значение поля Assembly name (Имя сборки) на `MDIEditor`.
4. Щелкните на кнопке Assembly Information... (Сведения о сборке...).
5. Измените информацию в полях Title (Заголовок), Description (Описание), Company (Компания), Product (Продукт) и Copyright (Авторские права), как показано на рис. 17.2.
6. Скомпонуйте проект, выбрав пункт меню Build⇨Build Solution (Компоновка⇨Компоновка решения).

Описание работы

Имя сборки определяет название сборки, которая была создана в процессе компоновки. Эта сборка должна быть развернута во время установки приложения. Свойства, измененные через диалоговое окно Assembly Information, модифицируют атрибуты сборки в файле `AssemblyInfo.cs`. Эта информация метаданных используется средствами развертывания. Информацию метаданных можно прочитать также из проводника Windows, выбирая исполняемый файл и щелкая на пункте меню Properties (Свойства). Вкладка Details (Сведения) позволяет просмотреть добавленную информацию.

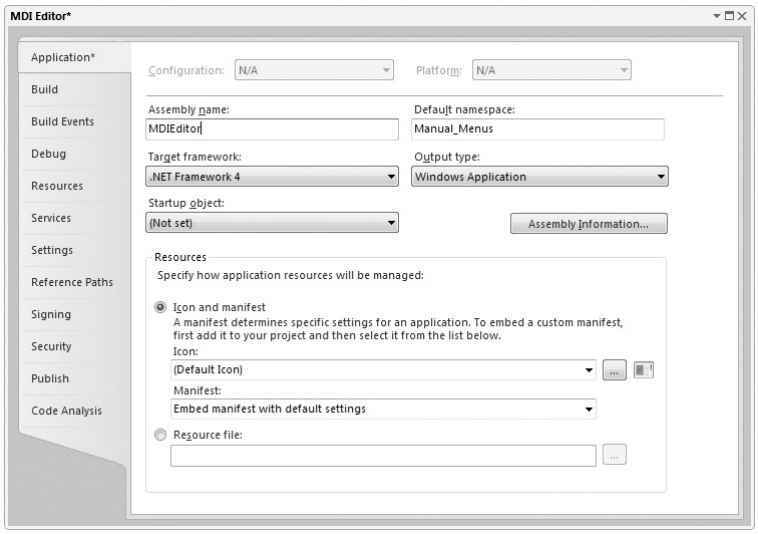


Рис. 17.1. *Вкладка* Application *окна свойств проекта*

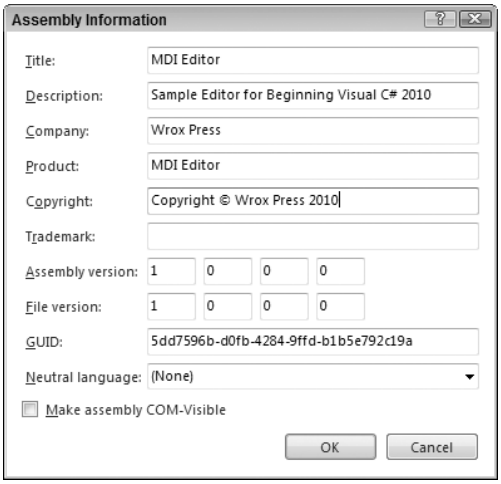


Рис. 17.2. *Установка сведений о сборке*

Для успешного развертывания сборки по сети требуется манифест, подписанный с помощью сертификата. Сертификат используется для отображения пользователю информации об организации, которая создала приложение. В результате пользователь может решить, можно ли доверять данному пакету развертывания. В следующем практическом занятии мы создадим сертификат, связанный с манифестами развертывания ClickOnce.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Подписание манифестов ClickOnce

1. В окне Solution Explorer выберите элемент Properties (Свойства) для проекта, а затем перейдите на вкладку Signing (Подписание), показанную на рис. 17.3.

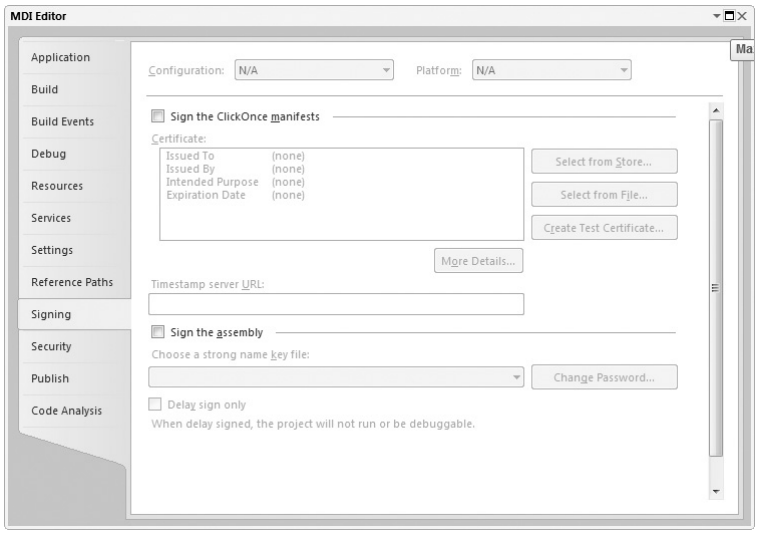


Рис. 17.3. Вкладка Signing окна свойств проекта

2. Отметьте флажок Sign the ClickOnce Manifests (Подписать манифесты ClickOnce).
3. Щелкните на кнопке Create Test Certificate... (Создать тестовый сертификат...), чтобы создать тестовый сертификат, связанный с манифестами ClickOnce. В ответ на запрос введите пароль для сертификата. Этот пароль необходимо запомнить для выполнения последующих настроек. Затем щелкните на кнопке OK.
4. Щелкните на кнопке More Details (Дополнительные сведения), чтобы ознакомиться с информацией о сертификате (рис. 17.4).

Рис. 17.4. *Просмотр информации о сертификате*

Описание работы

Сертификат предназначен для того, чтобы пользователь, устанавливающий приложение, мог идентифицировать создателя установочного пакета. Ознакомившись с сертификатом, пользователи могут решать, можно ли доверять данному пакету установки с точки зрения требований безопасности.

Только что созданный тестовый сертификат не предоставляет пользователю реальной информации о степени доверия, и, как будет показано далее, при его применении пользователь получает предупреждение о том, что ему нельзя доверять. Такой сертификат предназначен только для тестирования. Прежде чем приложение будет готово к развертыванию, необходимо получить реальный сертификат от такого центра сертификации, как VeriSign. Если приложение развертывается только внутри локальной сети организации, сертификат можно получить также с локального сервера сертификации, если таковой установлен. Сервер сертификации Microsoft Certificate Server может быть установлен в системе Windows Server 2003 или Windows Server 2008. При наличии такого сертификата, его можно конфигурировать, щелкая на кнопке Select from File (Выбрать из файла) на вкладке Signing.

В следующем практическом занятии мы сконфигурируем требования к безопасности сборки. При установке сборки в клиентской системе необходимо определить нужный уровень доверия.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Определение требований безопасности

1. В окне Solution Explorer выберите элемент Properties для проекта, перейдите на вкладку Security (Безопасность), показанную на рис. 17.5, а затем установите флажок Enable ClickOnce Security Settings (Включить параметры безопасности ClickOnce). Оставьте выбранную по умолчанию конфигурацию приложения, заслуживающего полного доверия, без изменений.

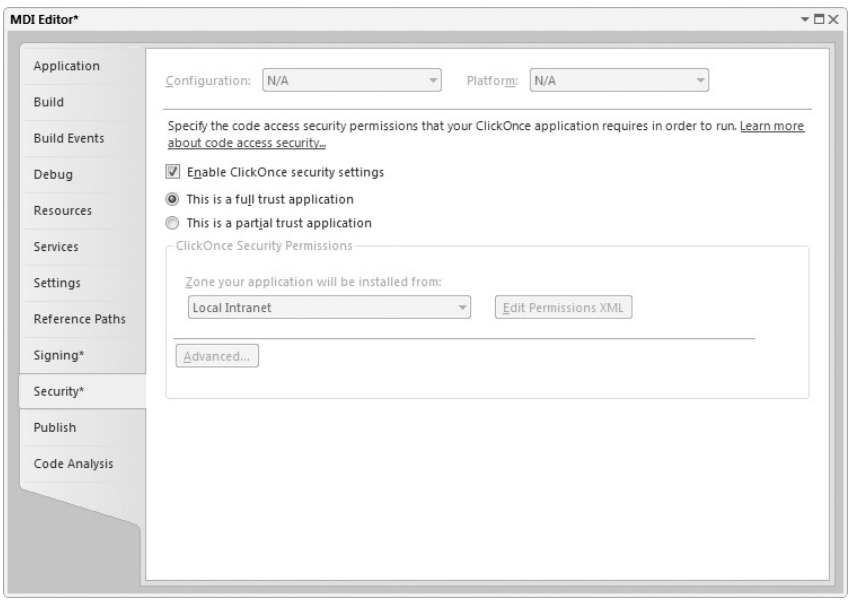


Рис. 17.5. *Вкладка* Security *окна свойств проекта*

Описание работы

Параметры настройки ClickOnce позволяют конфигурировать приложение так, чтобы оно требовало полного доверия либо выполнялось с частичным доверием внутри песочницы. При полном доверии приложение получает полный доступ к системе и может выполнять все, что разрешает запустивший его пользователь. При установке приложения пользователь получает предупреждение об этих требованиях. Приложение с частичным уровнем доверия не имеет доступа к другим разделам файловой системы кроме изолированного хранилища или изолированного раздела системного реестра. Приложение выполняется в режиме песочницы. Поскольку приложению MDI-редактора требуется доступ к файловой системе, для него должно быть установлено полное доверие.

После того как требования безопасности определены, можно приступать к публикации приложения, создав манифест развертывания. Это легко выполнить с помощью мастера публикации Publish Wizard, как показано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Дополнительные параметры конфигурирования публикации

1. Перейдите на вкладку Publish (Публикация) в окне свойств проекта. Щелкните на кнопке Options (Параметры), чтобы открыть диалоговое окно Publish Options (Параметры публикации), показанное на рис. 17.6. В списке в левой части диалогового окна выберите элемент Description (Описание). Введите название издателя (Publisher name), имя пакета (Suite name), название программного продукта (Product name) и URL-адрес сайта поддержки (Support URL).
2. Сконфигурируйте параметры обновления, щелкнув на кнопке Updates (Обновления) и отметив флажок The Application Should Check for Updates (Приложение должно проверять наличие обновлений), как показано на рис. 17.7.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование мастера публикации

1. Запустите мастер публикации (Publish Wizard), выбрав пункт меню Build⇨Publish SimpleEditor (Компоновка⇨Опубликовать SimpleEditor). Введите путь к веб-сайту `http://localhost/MDIEditor`, как показано на рис. 17.8. Щелкните на кнопке Next (Далее).

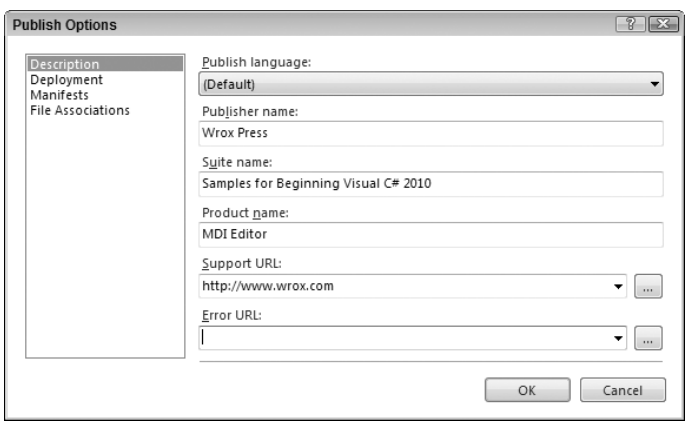


Рис. 17.6. *Диалоговое окно* Publish Options

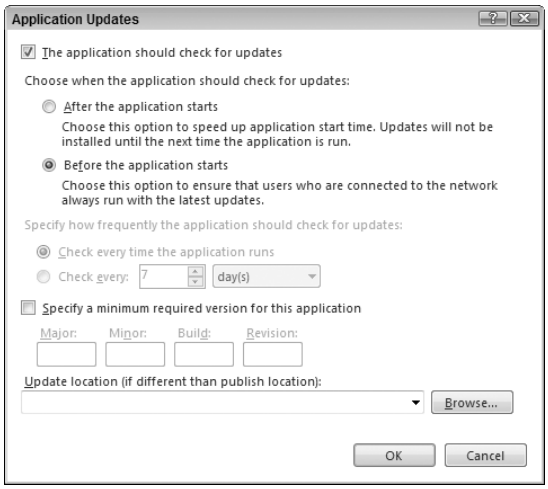


Рис. 17.7. *Установка флажка* The Application Should Check for Updates

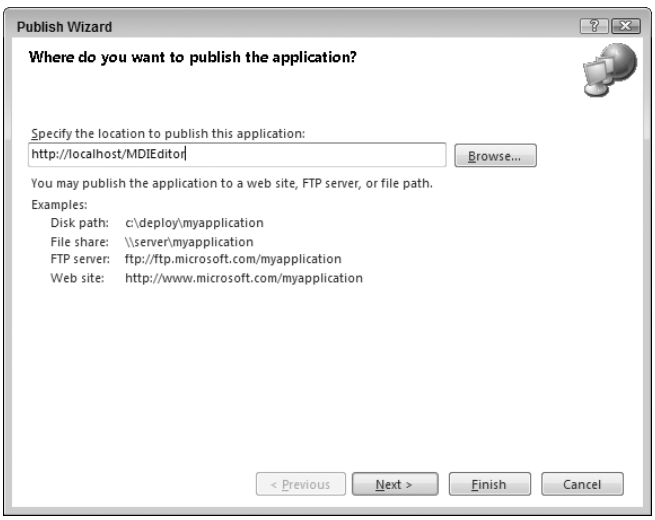


Рис. 17.8. *Указание пути к веб-сайту в мастере публикации*

Чтобы опубликовать приложение на веб-сервере в системе Windows 7 или Windows Vista, среда разработки Visual Studio 2010 должна быть запущена в расширенном режиме с административными привилегиями. Кроме того, должен быть установлен сервер Internet Information Server (IIS). Если IIS не установлен, выберите публикацию в локальной файловой системе.

2. На втором шаге мастера Publish Wizard выберите переключатель Yes, this application will be available online or offline (Да, это приложение будет доступно по сети или автономно), как показано на рис. 17.9. Щелкните на кнопке Next.
3. Последнее диалоговое окно отображает итоговую информацию, поскольку теперь мы готовы к выполнению публикации (рис. 17.10). Щелкните на кнопке Finish (Готово).

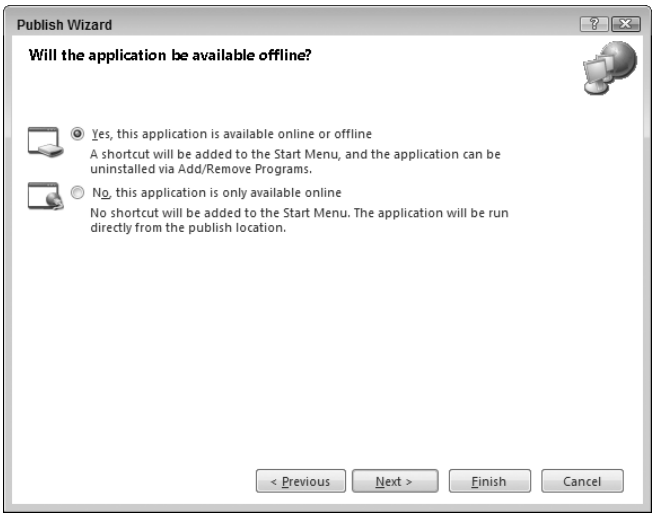


Рис. 17.9. *Второй шаг мастера публикации*

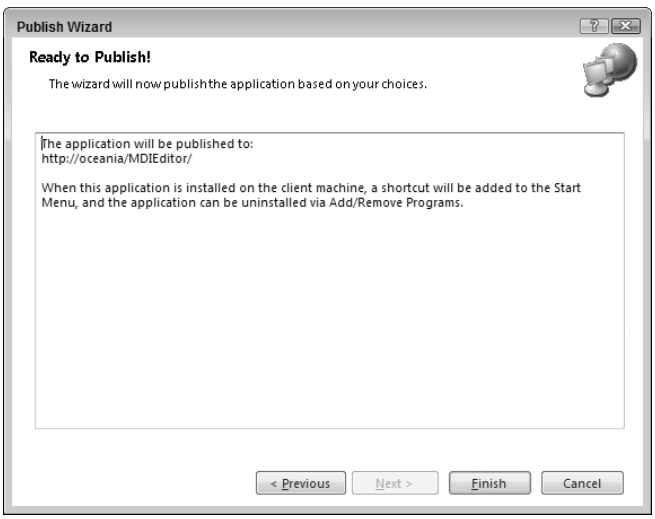


Рис. 17.10*. Итоговая информация в мастере публикации*

Описание работы

Мастер публикации создает веб-сайт на локальном веб-сервере IIS. Сборки приложений (исполняемые файлы и библиотеки), а также манифесты приложения и развертывания, файл `setup.exe` и образец веб-страницы, `publish.htm`, копируются на веб-сервер. Как показано в этом разделе, манифест развертывания описывает информацию установки. В Visual Studio манифест развертывания можно загрузить, открыв файл `MDIEditor.application` в окне Solution Explorer. Связь с манифестом приложения описана с помощью XML-элемента `<dependentAssembly>`:

```
<deployment install="true" mapFileExtensions="true">
  <subscription>
    <update>
      <beforeApplicationStartup />
    </update>
  </subscription>
  <deploymentProvider codebase="http://oceania/MDIEditor/MDIEditor.application" />
</deployment>
<compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2">
  <framework targetVersion="4.0" profile="Full" supportedRuntime="4.0.21205" />
</compatibleFrameworks>
<dependency>
  <dependentAssembly dependencyType="install"
      codebase="Application Files\MDIEditor_1_0_0_0\MDIEditor.exe.manifest"
      size="7416">
    <assemblyIdentity name="MDIEditor.exe" version="1.0.0.0"
      publicKeyToken="4e48aff44fcfc18a" language="neutral"
      processorArchitecture="x86" type="win32" />
    <hash>
      <dsig:Transforms>
        <dsig:Transform
          Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
      </dsig:Transforms>
      <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <dsig:DigestValue>+fiBvjYoMSuDkHZ680iLW2P4y+g=</dsig:DigestValue>
    </hash>
  </dependentAssembly>
</dependency>
```

За счет выбора опции, как показано на рис. 17.9 , мы указываем, что приложение будет доступно по сети и автономно. В результате приложение устанавливается в клиентской системе и становится доступным в меню Start. Кроме того, пользователи смогут удалять приложение, используя ярлык Add/Remove Programs панели инструментов. Если же вместо этого указать, что приложение должно быть доступно только по сети, для его загрузки с сервера и локального запуска пользователям придется всегда щелкать на ссылке веб-сайта.

Файлы, принадлежащие приложению, определяются выводом проекта. Чтобы увидеть файлы приложения на вкладке Publish его диалогового окна Properties, щелкните на кнопке Application Files (Файлы приложения). Откроется диалоговое окно Application Files (рис. 17.11). По умолчанию выполняется развертывание сборки и файла манифеста приложения.

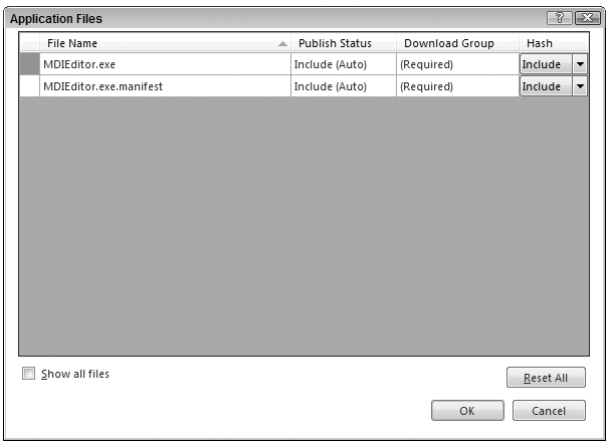


***Рис. 17.11**. Диалоговое окно Application Files*

Компоненты, необходимые для работы приложения, определяются в диалоговом окне Prerequisites (Необходимые компоненты), которое показано на рис. 17.12 и доступно посредством щелчка на кнопке Prerequisites (Необходимые компоненты). Как видно на рисунке, для приложений .NET 4 необходимый компонент .NET Framework 4 обнаруживается автоматически. Это диалоговое окно позволяет выбирать и другие необходимые компоненты.

Для установки приложений ClickOnce наличие административных привилегий не обязательно. Однако если необходимые компоненты не установлены в клиентской системе, для их установки требуются административные привилегии.

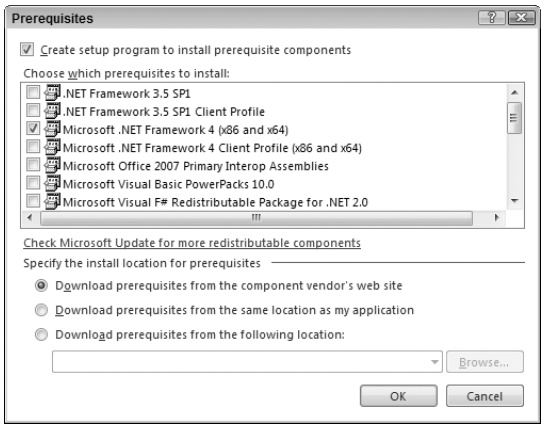


***Рис. 17.12**. Диалоговое окно* Prerequisites

Установка приложения с помощью технологии ClickOnce

Теперь можно установить приложение, выполнив действия, описанные в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Установка приложения MDI-редактора**

1. Откройте веб-страницу `publish.htm`, показанную на рис. 17.13.

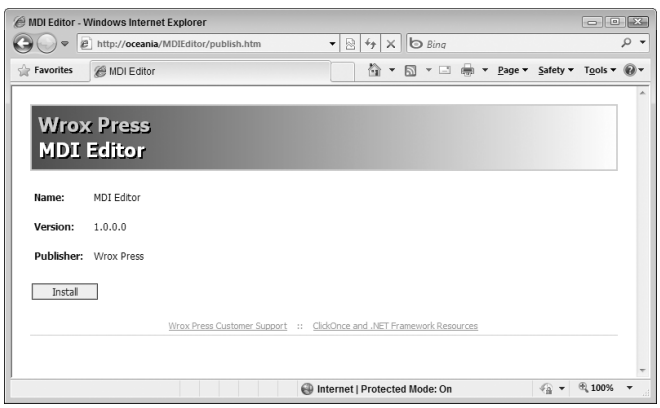


***Рис. 17.13**. Веб-страница* `publish.htm`

2. Щелкните на кнопке Install (Установить), чтобы установить приложение. Откроется окно с предупреждением безопасности, показанное на рис. 17.14.

***Рис. 17.14**. Окно с предупреждением безопасности*

3. Щелкните на ссылке More Information... (Дополнительные сведения...), чтобы ознакомиться с проблемами безопасности, связанными с приложением. Прочтите категории предупреждений этого диалогового окна, показанного на рис. 17.15.

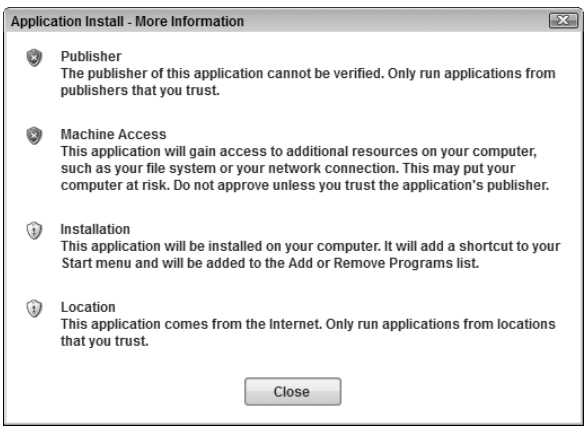


***Рис. 17.15**. Просмотр дополнительных сведений, связанных с безопасностью*

4. После ознакомления с информацией в этом диалоговом окне щелкните на кнопке Close (Закрыть), а затем на кнопке Install диалогового окна Application Install (Установка приложения), если доверяете созданному приложению.

Описание работы

При открытии файла `publish.htm` целевое приложение проверяется на предмет наличия исполняющий среды .NET версии 4. Эта проверка выполняется JavaScript-функцией, встроенной в HTML-страницу. Если исполняющая среда отсутствует, она устанавливается перед установкой клиентского приложения. При использовании настроек публикации, установленных по умолчанию, исполняющая среда копируется из сайта Microsoft.

Щелчок на ссылке установки приложения ведет к открытию манифеста развертывания для установки приложения. Затем пользователь информируется о любых возможных проблемах безопасности, связанных с приложением. Если пользователь щелкает на кнопке OK, приложение устанавливается.

Создание и использование обновлений приложения

При использовании ранее сконфигурированных параметров обновления клиентское приложение автоматически осуществляет проверку наличия новой версии на веб-сервере. В следующем практическом занятии мы рассмотрим такой сценарий на примере приложения `MDI Editor`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Обновление приложения

1. Внесите в приложение `MDI Editor` изменения, которые могут быть замечены немедленно — такие как смена цвета фона поля форматированного текста, определенного в файле `frmEditor.cs`.
2. Откройте раздел Publish в окне свойств проекта и удостоверьтесь, что номер версии публикации изменился на новый.
3. Скомпонуйте приложение и щелкните на кнопке Publish Now (Опубликовать немедленно) в разделе Publish окна свойств проекта.
4. Не щелкайте на ссылке `publish.htm` веб-страницы; вместо этого запустите приложение из меню Start. После запуска приложения откроется диалоговое окно Update Available (Доступно обновление), показанное на рис. 17.16 и содержащее запрос о необходимости загрузки новой версии. Щелкните на кнопке OK, чтобы загрузить новую версию. После запуска новой версии приложения его поле форматированного текста будет окрашено в выбранный цвет.

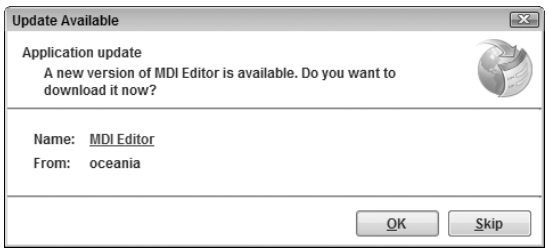


Рис. 17.16. *Диалоговое окно* Update Available

Описание работы

Политика обновления определяется XML-элементом `<update>` в манифесте развертывания. Ее можно изменить, щелкая на кнопке Updates на вкладке настроек Publish. Не забудьте, что доступ к настройкам Publish открывается из окна свойств проекта. Диалоговое окно Application Updates (Обновление приложения) показано на рис. 17.17.

Используйте это диалоговое окно для указания того, должно ли клиентское приложение вообще искать доступные обновления. Если проверка наличия обновлений требуется, можно определить, должна ли проверка выполняться перед запуском приложения или в фоновом режиме во время его работы. Если обновление должно выполняться в фоновом режиме, можно указать временной интервал между обновлениями: с каждым запуском приложения или через указанное количество часов, дней или недель.

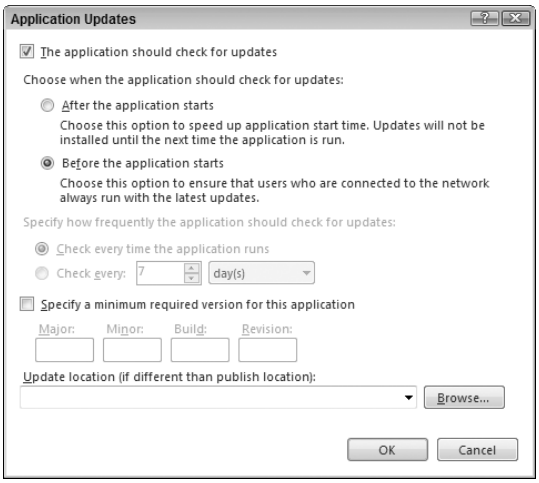


Рис. 17.17. *Диалоговое окно* Application Updates

Типы проектов установки и развертывания Visual Studio

Откройте диалоговое окно Add New Project (Добавить новый проект). Откройте меню Other Project Types⇨Visual Studio Installer (Другие типы проектов⇨Программа установки Visual Studio) и в панели Installed Templates (Установленные шаблоны) выберите шаблон Setup and Deployment (Установка и развертывание). Окно будет иметь вид, показанный на рис. 17.18.

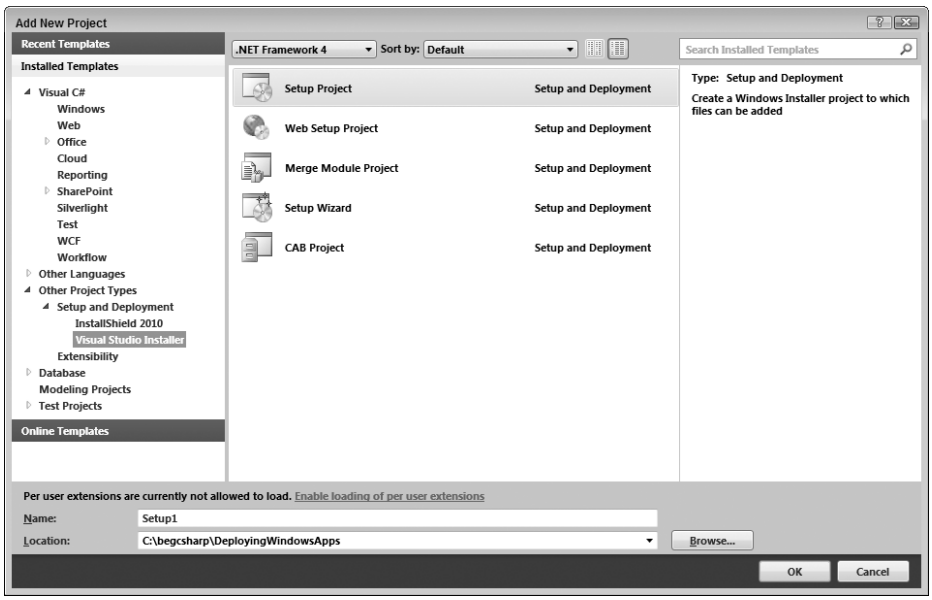


Рис. 17.18. *Диалоговое окно* Add New Project

Типы проектов и предоставляемые ими возможности описаны в следующем списке.

- Шаблон Setup Project (Проект установки) — тот, который мы будем использовать. Этот шаблон применяется для создания пакетов программы установки Windows, поэтому его можно использовать для развертывания Windows-приложений.
- Шаблон Web Setup Project (Проект веб-установки) можно использовать для установки веб-приложений. Этот шаблон проекта применяется в главе 20.
- Шаблон Merge Module Project (Проект модуля слияния) предназначен для создания модулей слияния для программы установки Windows. Модуль слияния — это файл программы установки, который может быть включен в несколько установочных пакетов для программы установки Windows. Модуль слияния можно создать для его включения в пакеты установки компонентов, которые должны устанавливаться более чем одной установочной программой. Одним из примеров модуля слияния служит сама исполняющая среда .NET. Она поставляется в модуле слияния, поэтому может быть включена в пакет программы установки приложения. Мы используем модуль слияния в примере приложения.
- Мастер установки (Setup Wizard) — пошаговое средство выбора других шаблонов. Прежде всего, необходимо ответить на вопрос, нужно создать программу установки приложения или же распространяемый пакет. В зависимости от выбора будет создан пакет программы установки Windows, модуль слияния или CAB-файл.
- Шаблон Cab Project (Проект архивного файла) позволяет создавать архивные (CAB) файлы. CAB-файлы можно использовать для слияния нескольких сборок в один файл и его сжатия. Поскольку CAB-файлы могут быть сжаты, веб-клиент получает возможность загрузки с сервера файла меньшего размера.

Архитектура программы установки Microsoft Windows

До появления программы установки Windows программистам приходилось создавать специализированные программы установки. Мало того, что построение таких установочных программ было не только более трудоемким, так еще и многие из них не соответствовали правилам Windows. Часто системные DLL-библиотеки оказывались замещенными более старыми версиями, поскольку программа установки не выполняла проверку версий. Кроме того, часто каталог, в который копировались файлы приложения, выбирался неправильно. Например, если при использовании жестко закодированной строки каталога, такой как `C:\Program Files`, системный администратор изменял применяемую по умолчанию букву диска, или установка выполнялась в интернациональной версии операционной системы, в которой этот каталог имел иное имя, установка оказывалась неудачной.

Первая версия программы установки Windows была реализована в качестве составной части пакета Microsoft Office 2000 и виде дистрибутивного пакета, который мог включаться в пакеты других приложений. В эту первую версию была добавлена поддержка для регистрации компонентов COM+. В версию 1.2 была добавлена поддержка механизма защиты файлов Windows ME. Версия 2.0 была первой версией, которая включала поддержку для установки сборок .NET, а также поддержку 64-разрядной версии Windows. Минимальная версия программы установки Windows, которая может быть использована в среде .NET 4 — версия 3.1

Термины программы установки Windows

Для успешной работы с программой установки Windows нужно ознакомиться с некоторыми терминами, применяемыми при описании ее технологии: пакетами, функциональными средствами и компонентами.

В контексте программы установки Windows понятие компонента отличается от компонента в контексте .NET Framework. Компонент программы установки Windows — это всего лишь отдельный файл (или несколько файлов, логически связанных друг с другом). Таким файлом может быть исполняемый файл, DLL-библиотека или даже простой текстовый файл.

Как видно на рис. 17.19, *пакет* состоит из одного или более функциональных средств. Пакет — это отдельная база данных Microsoft Installer (Программа установки Microsoft) или MSI. *Функциональное средство* — это возможности продукта с точки зрения пользователя, и оно может состоять из других функциональных средств и компонентов. *Компонент* представляет взгляд разработчика на установку. Это наименьший модуль установки, состоящий из одного или более файлов. Следует различать функциональные средства и компоненты, поскольку один компонент может быть включен в несколько функциональных средств (как компонент 2 на рисунке). Одно функциональное средство не может входить в состав нескольких других функциональных средств.

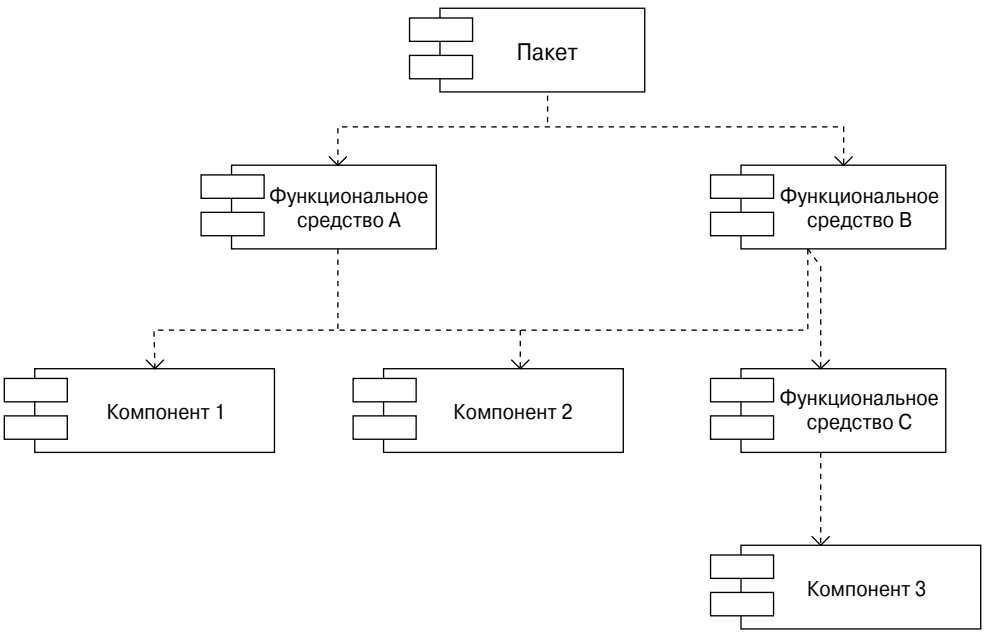


***Рис. 17.19**. Состав пакета*

Рассмотрим реальный пример функциональных средств, которым вы уже должны располагать: Visual Studio 2010. С помощью опции Programs and Features (Программы и компоненты) панели управления можно изменять установленные функциональные средства Visual Studio после установки, щелкая на кнопке Uninstall/Change (Удалить/Изменить), как показано на рис. 17.20.

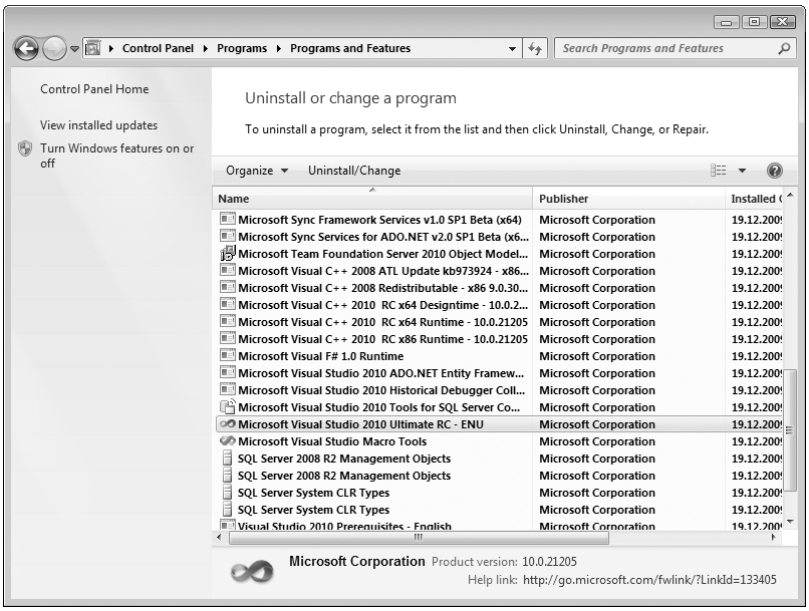


***Рис. 17.20**. Опция* Programs and Features *панели управления*

Щелкая на кнопке Uninstall/Change, можно открыть мастер обслуживания (Maintenance Wizard) Visual Studio 2010. Это хороший способ увидеть функциональные средства в действии. Щелкая на знаках плюса и минуса в отображаемой слева панели древовидной структуры, можно просмотреть все функциональные средства пакета Visual Studio 2010 (рис. 17.21).

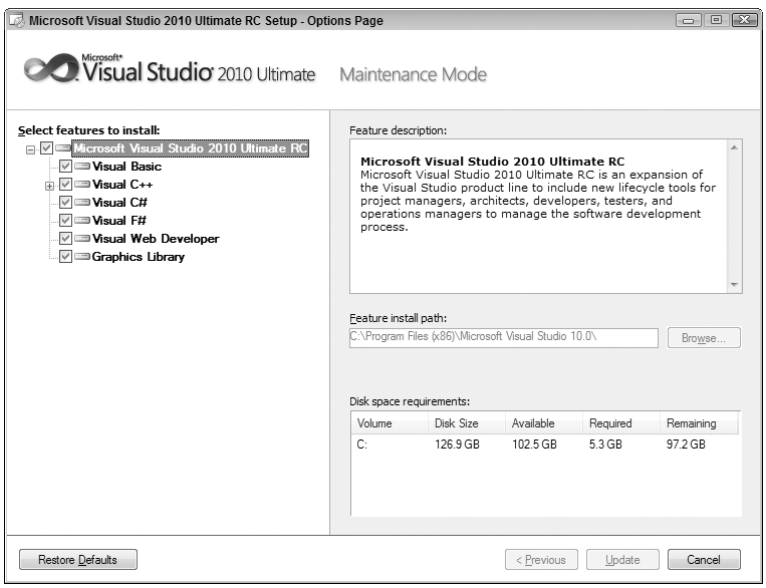


***Рис. 17.21**. Функциональные средства пакета Visual Studio 2010*

Пакет Visual Studio 2010 включает в себя следующие функциональные средства: Visual Basic, Visual C++, Visual C#, Visual F#, Visual Web Developer и Graphics Library.

Преимущества программы установки Windows

Программа установки Windows обеспечивает несколько преимуществ.

- Функциональное средство может быть установлено, не установлено или объявлено. При использовании функции *объявления* пакет устанавливается при первом использовании. Возможно, у вас уже возникала ситуация, когда программа установки Windows запускалась при работе с Microsoft Word. При использовании еще не установленного объявленного функционального средства Word оно будет автоматически установлено при первом же обращении к нему.
- В случае повреждения приложения оно может быть *самостоятельно восстановлено* с помощью функции восстановления пакетов программы установки Windows.
- В случае неудачи установки программа установки выполнит автоматический *откат*. После неудачной установки все остается в том же состоянии, каком пребывало до ее запуска: в системе не остается никаких дополнительных записей системного реестра, файлов и т.п.
- Функция *отмены установки* удаляет все связанные с ней файлы, записи системного реестра и т.п. — другими словами, приложение может быть полностью удалено. Из системы удаляются все временные файлы, а системный реестр восстанавливается.

Для получения информации о скопированных файлах и записях системного реестра можно просмотреть таблицы файла базы данных MSI.

Создание установочного пакета для приложения `MDI Editor`

В этом разделе мы используем созданное в главе 16 решение для приложения `MDI Editor`, чтобы средствами Visual Studio 2010 создать пакет программы установки Windows. Естественно, при выполнении описанных действий можно использовать любое другое разработанное приложение Windows Forms или WPF. Нужно только изменить некоторые из используемых имен.

Планирование установки

Прежде чем приступить к построению программы установки, необходимо спланировать ее содержимое. Продумайте ответы на перечисленные ниже вопросы.

- **Какие файлы требуются для приложения?** Естественно, приложению требуется исполняемый файл и, возможно, сборки определенных компонентов. Нам не нужно будет определять все зависимости между этими элементами, поскольку они включаются автоматически. Другими необходимыми файлами в числе прочих могут быть файл документации, файл `readme.txt`, файл лицензионного соглашения, шаблон документа, рисунки и файлы конфигурации. Иначе говоря, должны быть известны все необходимые файлы.

  Для приложения `MDI Editor`, разработанного в главе 16, требуется исполняемый файл, а также файлы `readme.rtf` и `license.rtf` и растровое изображение из сайта Wrox Press, которое будет отображаться в диалоговых окнах установки.

- **Какие каталоги должны использоваться?** Установка файлов приложения должна выполняться в каталог `Program Files\`*`Имя_приложения`*. Имя каталога `Program Files`

зависит от языкового варианта операционной системы. Кроме того, администратор может выбирать другие пути для установки данного приложения. Точное знание местоположения этого каталога не обязательно, поскольку для его нахождения существует специальная функция API-интерфейса. Программа установки позволяет помещать файлы в специальную заранее определенную папку в каталоге `Program Files`.

Стоит еще раз подчеркнуть: ни при каких обстоятельствах нельзя жестко кодировать каталоги! В различных международных версиях ОС эти каталоги имеют различные имена. Даже если приложение поддерживает только англоязычную версию Windows (что не слишком разумно), не исключено, что системный администратор переместил эти каталоги на другие диски.

Исполняемый файл приложения `MDI Editor` будет установлен в заданный по молчанию каталог приложения, если только пользователь не выберет другой путь.

- **Как пользователь должен получать доступ к приложению?** Например, можно добавить команду быстрого доступа к исполняемому файлу в меню Start либо поместить соответствующий значок на рабочий стол. Если хотите поместить значок на рабочий стол, удостоверьтесь, что это устраивает пользователя. В соответствии с рекомендациями по Windows XP рабочий стол должен оставаться как можно более свободным. В среде Windows 7 пользователи могут помещать на рабочий стол гаджеты (небольшие активные программы). Это одна из причин того, что рабочий стол должен оставаться свободным, чтобы пользователь имел возможность расположить значки и гаджеты удобным для него образом. Поэтому приложение `MDI Editor` должно быть доступно из меню Start.
- **Что служит носителем для распространения?** Должны ли пакеты установки быть помещены на компакт-диск, гибкие диски или в сетевой каталог совместного использования?
- **На какие вопросы должны ответить пользователи?** Должны ли они принять условия лицензионного соглашения, прочесть файл ReadMe или ввести путь для установки? Требуются ли для установки какие-то дополнительные опции?

  Диалоговые окна, по умолчанию предоставляемые программой установки Visual Studio 2010, вполне достаточны для проекта программы установки Windows, который будет создан в оставшейся части этой главы. Мы запросим каталог, в который должна быть установлена программа (пользователь может выбрать путь, отличающийся от заданного по умолчанию), отобразим файл ReadMe и предложим пользователю принять условия лицензионного соглашения.

Создание проекта

Теперь, когда состав пакета установки известен, программу установки Visual Studio 2010 можно использовать для создания проекта программы установки и добавления всех файлов, предназначенных для установки. В следующем практическом занятии для конфигурирования проекта применяется мастер Project Wizard (Мастер проекта).

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Создание проекта программы установки Windows**

1. Откройте файл решения проекта `MDI Editor`, созданный в главе 16. Проект установки будет добавлен в существующее решение. Если по какой-либо причине решение не было создано, скопируйте всю папку `MDI Editor` из кода примеров для данной главы. Откройте файл решения `MDI Editor.sln` в среде Visual Studio с помощью пункта меню File⇨Open⇨Project/Solution (Файл⇨Открыть⇨Проект/Решение).

2. Добавьте в решение проект установки **`MDIEditorSetup`**. Для этого откройте меню File⇨Add New Project (Файл⇨Добавить новый проект), выберите опцию меню Other Project Types⇨Setup and Deployment (Другие типы проектов⇨Установка и развертывание), выберите шаблон Setup Project (Проект установки), как показано на рис. 17.22, и щелкните на кнопке OK.

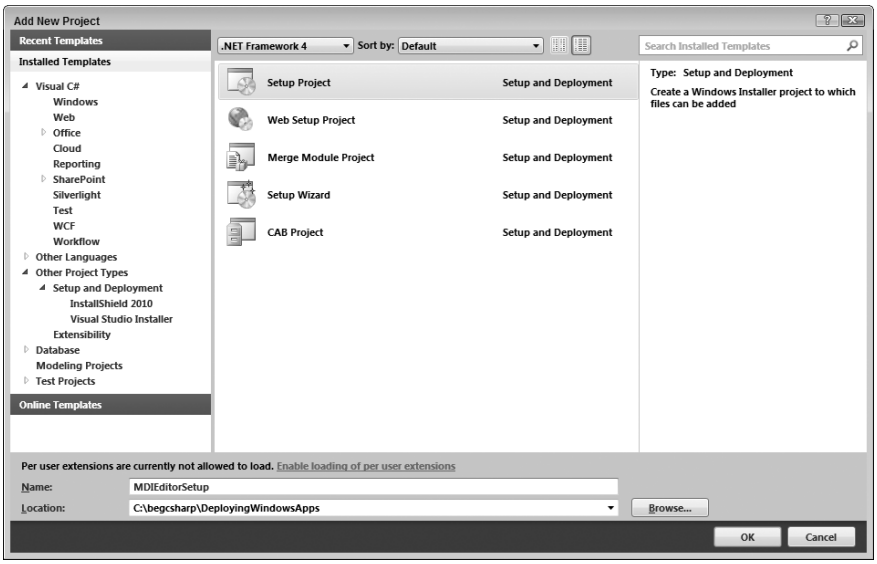


Рис. 17.22. *Выбор шаблона* Setup Project

Свойства проекта

На данный момент мы располагаем только файлом проекта для решения установки. Файлы, которые требуется установить, уже должны быть определены, но понадобится также сконфигурировать свойства проекта. Для этого необходимо разобраться с опциями Packaging (Упаковка) и Bootstrapper (Начальный загрузчик).

Упаковка

Установка запускается из MSI, но с помощью трех опций диалогового окна, показанного на рис. 17.23, можно определить способ упаковки файлов, которые предназначены для установки. Это диалоговое окно открывается посредством щелчка правой кнопки мыши на проекте `MDIEditorSetup` и выбора в контекстном меню пункта Properties.

Вначале рассмотрим опции раскрывающегося списка Package Files (Упаковать файлы).

- As Loose Uncompressed Files (В виде отдельных несжатых файлов). Сохраняет все файлы программы и данных в их первоначальном виде. Никакое сжатие не выполняется.
- In Setup File (В файл установки). Объединяет и сжимает все файлы в один файл MSI. Эта опция может быть замещена для отдельных компонентов в пакете. При помещении всех файлов в один файл MSI необходимо убедиться, что размер программы установки соответствует предполагаемому целевому носителю, такому как компакт-диски или гибкие диски. Если общий размер предназначенных для установки файлов превышает емкость одного гибкого диска, можно попробовать изменить параметр сжатия, выбирая опцию Optimized for Size (Оптимизировать для размера) из рас-

крывающегося списка Compression (Сжатие). Если файлы по-прежнему не умещаются на выбранном носителе, выберите для упаковки следующую опцию.

- In cabinet file(s) (В архивный файл(ы)). При использовании этого метода файл MSI служит только для загрузки и установки CAB-файлов. Применение CAB-файлов позволяет устанавливать размеры файлов так, чтобы установку можно было выполнять с компакт-дисков или гибких дисков (для выполнения установки с гибких дисков размеры файлов можно устанавливать равными 1440 Кбайт).

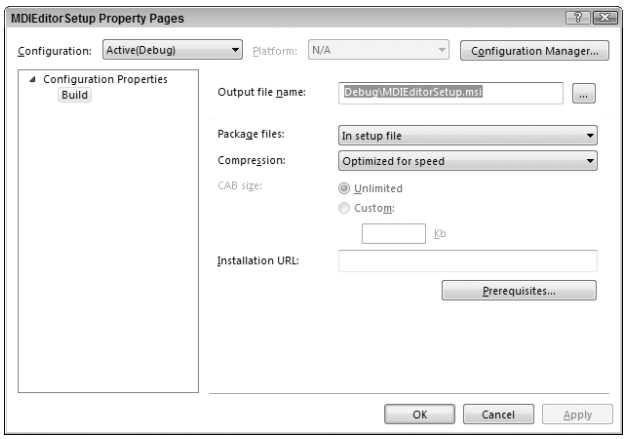


***Рис. 17.23**. Окно свойств проекта* `MDIEditorSetup`

Необходимые компоненты

В этом же диалоговом окне можно конфигурировать необходимые компоненты, которые должны присутствовать в системе, прежде чем приложение сможет быть установлено. Щелчок на кнопке Settings (Настройки), расположенной рядом с текстовым полем Prerequisites URL (URL-адрес необходимых компонентов) ведет к открытию диалогового окна Prerequisites (Необходимые компоненты), показанного на рис. 17.24.

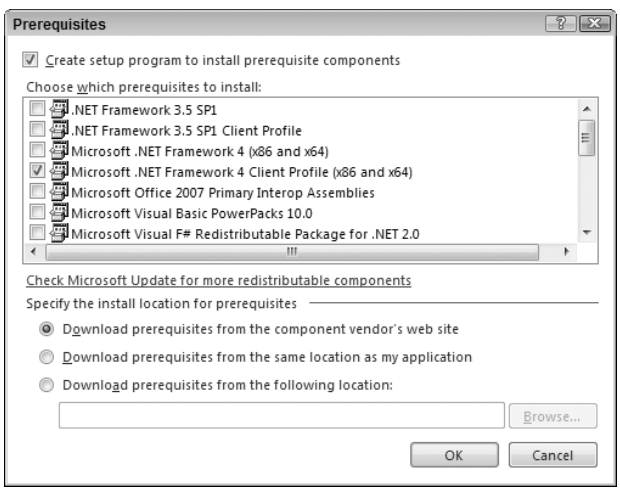


***Рис. 17.24**. Диалоговое окно* Prerequisites

Как видите, компонент .NET Framework 4 Client Profile (Клиентский профиль .NET Framework 4) выбран в качестве необходимого по умолчанию. Если клиентская система не имеет установленной версии .NET Framework, она будет установлена из программы установки. Как видно из следующего перечня, можно выбирать также и другие необходимые компоненты.

- Windows Installer 3.1. Windows Installer 3.1 требуется для пакетов программ установки, созданных с помощью Visual Studio 2010. Если целевой системой является Windows Vista или Windows Server 2008, программа установки уже присутствует в ней. При использовании более ранних систем нужная версия программы установки Windows может отсутствовать, поэтому данную опцию можно выбрать, чтобы включить Windows Installer 3.1 в пакет программы установки. В Windows 7 используется версия 5 программы установки Windows, а в Visual Studio 2010 — версия 4.5.
- SQL Server 2008 Express. Если нужно, чтобы база данных была установлена в клиентской системе, в пакет программы установки можно включить компонент SQL Server 2008 Express Edition. Доступ к серверу SQL Server посредством ADO.NET описан в главе 24.
- Microsoft Office 2007 Primary Interop Assemblies (Основные сборки взаимодействия Microsoft Office 2007). Этот компонент позволяет устанавливать основные сборки взаимодействия с Office 2007 для тех приложений, которые используют средства автоматизации офисных операций.
- Visual Basic PowerPacks 10.0. Этот компонент предоставляет дополнительные средства программирования на языке Visual Basic, которые могут быть установлены вместе с ним.
- Visual F# Redistributable Package (Распространяемый пакет Visual F#). Этот пакет нужен для приложений, которые написаны на языке программирования F#.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Конфигурирование проекта

1. Измените рассмотренную опцию Prerequisites на странице Property, включив компонент Windows Installer 3.1, чтобы обеспечить возможность установки приложения в системах, в которых программа установки Windows Installer 3.1 не доступна. Измените также имя выходного файла на `WroxMDIEditor.msi`, как показано на рис. 17.25. Затем щелкните на кнопке OK.

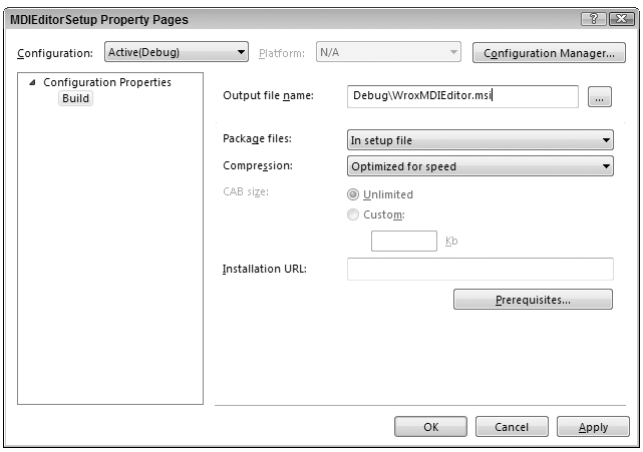


***Рис. 17.25**. Изменение имени выходного файла*

2. В окне Properties установите значения свойств проекта, как указано в следующей таблице:

Свойство	Значение
Author	Wrox Press
Description	MDI Editor to print and edit text files.
Keywords	Installer, Wrox Press, MDI Editor
InstallAllUsers	True
Manufacturer	Wrox Press
ManufacturerUrl	http://www.wrox.com
Product Name	Wrox MDI Editor
SupportUrl	http://p2p.wrox.com
Title	Installation Demo for MDI Editor
Version	1.0.0

Редакторы установки

Среда Visual Studio 2010 предоставляет шесть редакторов для проекта установки. Нужный редактор выбирается с помощью пункта меню View⇨Editor (Вид⇨Редактор) в открытом проекте развертывания.

- Редактор File System Editor (Редактор файловой системы) используется для добавления файлов в пакет установки.
- Редактор Registry Editor (Редактор реестра) позволяет создавать записи системного реестра для приложения.
- Редактор File Types Editor (Редактор типов файлов) позволяет регистрировать для приложения конкретные расширения файлов.
- Редактор User Interface Editor (Редактор интерфейса пользователя) позволяет добавлять и конфигурировать диалоговые окна, отображаемые во время установки программного продукта.
- Редактор Custom Actions Editor (Редактор пользовательских действий) позволяет запускать пользовательские программы во время установки и удаления.
- Редактор Launch Conditions Editor (Редактор условий запуска) позволяет указывать требования к приложению — например, необходимость наличия исполняющей среды .NET.

Редактор File System Editor

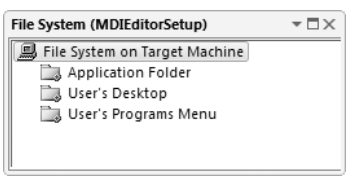


***Рис. 17.26**. Предварительно определенные папки*

Редактор File System Editor позволяет добавлять файлы в пакет установки и конфигурировать каталоги, в которые они должны быть установлены. Для открытия этого редактора выберите пункт меню View⇨Editor⇨File System (Вид⇨Редактор⇨Файловая система). При этом автоматически открывается ряд предварительно определенных специальных папок, как показано на рис. 17.26.

- Папка `Application Folder` служит для хранения исполняемых файлов и библиотек. Ее местоположение

определяется как [*ПапкаProgramFiles*]\[*Изготовитель*]\[*Имя_продукта*]. В англоязычных системах вместо спецификатора [*ПапкаProgramFiles*] подставляется строка C:\Program Files. Каталоги [*Изготовитель*] и [*Имя_продукта*] определяются свойствами проекта Manufacturer и ProductName.

- Если хотите поместить значок на рабочий стол, можно использовать папку User's Desktop (Рабочий стол пользователя). По умолчанию путь к этой папке — C:\Users*имя-пользователя*\Desktop или C:\Users\All Users\Desktop, в зависимости от того, выполняется установка для одного или для всех пользователей.
- Обычно пользователь будет запускать программу из меню All Programs (Все программы). По умолчанию путь к этой папке — C:\Documents and Settings*имя_пользователя*\Start Menu\Programs. Ярлык приложения можно поместить в это меню. Имя ярлыка должно включать название компании и имя приложения, чтобы пользователь мог легко идентифицировать приложение (например, Microsoft Excel).

Некоторые приложения создают подменю, из которого можно запускать более одного приложения — например, Microsoft Visual Studio 2010. Как указано в рекомендациях Windows, многие программы делают это без веских причин, помещая в это меню программы, которые не нужны. Например, в эти меню не следует помещать программу удаления, поскольку данное функциональное средство доступно через значок Programs and Features (Программы и компоненты) панели управления и должно использоваться оттуда. Файл справки также не следует помещать в это меню, поскольку он должен быть доступен из приложения. Таким образом, для многих приложений достаточно поместить ярлык приложения непосредственно в меню All Programs. Цель перечисленных ограничений — предотвращение загромождения меню Start чересчур большим количеством элементов.

Прекрасную справочную информацию по этому вопросу можно найти в статье, посвященной спецификациям приложений для Microsoft Windows 7. Эти документы можно найти на веб-странице http://msdn.microsoft.com/en-us/windows/dd203105.aspx.

Дополнительные папки можно добавить, щелкая правой кнопкой мыши и выбирая в контекстном меню пункт Add Special Folder (Добавить специальную папку). Некоторые из этих папок перечислены ниже.

- Global Assembly Cache (GAC) Folder (Папка глобального кэша сборок) — это папка, в которой можно устанавливать сборки совместного использования. Глобальный кэш сборок (GAC) предназначен для сборок, которые должны совместно использоваться несколькими приложениями.
- User's Personal Data Folder (Папка персональных данных пользователя) — заданная по умолчанию папка для хранения документов пользователя. Путь к ней по умолчанию выглядит как C:\Users\[*имя_пользователя*]\My Documents. Это заданный по молчанию каталог, используемый Visual Studio для хранения проектов.
- Ярлыки, помещенные в меню User's Send To Menu (Меню "Отправить..." пользователя) расширяет контекстное меню Send To (Отправить) при наличии выбранного файла. Как правило, это контекстное меню позволяет пользователю отправить файл в соответствующее место назначения, такое как гибкий диск, адресат электронной почты или папка My Documents (Мои документы).

Добавление элементов в специальные папки

Элементы в специальную папку можно добавить, выбирая папку, а затем выбирая пункт меню Action⇨Add Special Folder (Действие⇨Добавить специальную папку). При этом можно выбирать опции Project Output (Вывод проекта), Folder (Папка), File (Файл) или Assembly (Сборка). Добавление вывода проекта ведет к автоматическому добавлению сгенерированных выходных файлов и файла .dll или .exe, в зависимости от того, является добавленный проект библиотекой компонентов или приложением. Выбор опции Project

Output либо Assembly ведет к автоматическому добавлению в папку всех зависимостей (всех сборок, на которые имеются ссылки).

Свойства файла

При выборе свойств файла в папке можно устанавливать свойства, описанные в табл. 17.1. В зависимости от типов файлов некоторые их этих свойств не применимы, а некоторые дополнительные свойства в таблице не показаны.

Таблица 17.1. Свойства файла, доступные для установки

Свойство	Описание
`Condition`	Это свойство позволяет определить условие, при котором выбранный файл должен быть установлен. Оно может быть полезно, если данный файл должен добавляться только для конкретной версии операционной системы, или если пользователь должен осуществить выбор в диалоговом окне
`Exclude`	Если установка данного файла не требуется, значение этого свойства следует установить равным `True`. В результате файл может оставаться в проекте, но не устанавливаться. Файл может быть исключен из установки при заведомом отсутствии какой-либо зависимости от него или если он уже присутствует в каждой системе, в которой приложение будет развертываться
`PackageAs`	Это свойство позволяет изменить заданный по умолчанию способ добавления файла в пакет программы установки. Например, если в конфигурации проекта способ добавления указан как In Setup File (В файл установки), это свойство позволяет для конкретного файла изменить способ добавления на Loose (Свободный), чтобы данный файл не добавлялся в файл базы данных MSI. Это полезно, например, если нужно добавить файл ReadMe, который пользователь должен прочесть до начала установки. Очевидно, что этот файл не должен сжиматься, даже если все остальные файлы сжаты
`Permanent`	Установка значения этого свойства равным `True` означает, что файл останется на целевом компьютере после удаления программного продукта. Это свойство можно использовать для файлов конфигурации. Вы могли встречаться с подобной ситуацией при установке новой версии Microsoft Outlook: при конфигурировании Microsoft Outlook, его удалении и последующей повторной установке. При этом необходимость повторного конфигурирования приложения отсутствует, поскольку конфигурация последней установки не удаляется
`ReadOnly`	Это свойство при установке устанавливает для файла атрибут “только для чтения”
`Vital`	Это свойство означает, что файл совершенно необходим для установки данного продукта. Если установка данного файла оказывается неудачной, вся установка отменяется, и программа установки выполняет откат.

В следующем практическом занятии мы добавим файлы, которые должны быть развернуты посредством пакета программы установки Windows.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Добавление файлов в пакет программы установки

1. Посредством пункта меню Project⇨Add⇨Project Output (Проект⇨Добавить⇨Вывод проекта) добавьте основной вывод проекта `MDI Editor` в папку `Application Folder` проекта программы установки. В диалоговом окне Add Project Output Group (Добавить группу вывода проекта) выберите опцию Primary Output (Основной вывод), как показано на рис. 17.27.

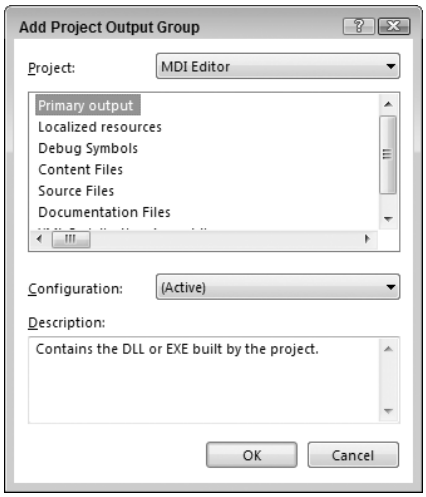


***Рис. 17.27**. Диалоговое окно Add Project Output Group*

Щелкните на кнопке OK, чтобы добавить основной вывод проекта `MDI Editor` в папку `Application Folder` автоматически открывающегося редактора File System Editor. В данном случае основным выводом является файл `MDIEditor.exe`.

2. Дополнительные файлы — файлы логотипа, лицензионного соглашения и ReadMe. В среде редактора File System Editor в папке `Application Folder` создайте подкаталог `Setup`. Для этого выберите папку `Application Folder`, а затем выберите пункт меню Action⇨Add⇨Folder (Действие⇨Добавить⇨Папку).

В Visual Studio меню Action доступно только при выборе каких-нибудь элементов в окнах редакторов установки. Если элемент выбран в окне Solution Explorer (Проводник решений) или Class View (Представление классов), меню Action не доступно.

3. Добавьте файлы `wroxlogo.bmp`, `wroxsetuplogo.bmp`, `readme.rtf` и `license.rtf` в папку `Setup`, щелкая на ней правой кнопкой мыши и выбирая пункт Add⇨File (Добавить⇨Файл) в контекстном меню. Эти файлы доступны в коде примеров для данной главы, но их легко создать и самостоятельно. Текстовые файлы можно заполнить информацией лицензионного соглашения и файла ReadMe. Свойства этих файлов, которые будут использованы в диалоговых окнах программы установки, можно не изменять.

   Размер растрового изображения, хранящегося в файле `wroxsetuplogo.bmp`, должен быть равным 500 пикселей по ширине и 70 пикселей по высоте. 420 левых пикселей растрового изображения должны содержать только фоновое изображение, поскольку текст диалоговых окон установки будет перекрывать эту область.

4. Добавьте файл `readme.txt` в папку `Application Folder`. Этот файл должен быть доступен для прочтения пользователями до начала установки. Установите значение свойства `PackageAs` в `vsdpaLoose`, чтобы предотвратить сжатие этого файла в пакет программы установки. Чтобы воспрепятствовать изменению этого файла, установите значение его свойства `ReadOnly` в `true`.

   Теперь проект содержит два файла ReadMe: `readme.txt` и `readme.rtf`. Файл `readme.txt` может быть прочтен пользователем, устанавливающим приложение, до

начала установки. Файл `readme.rtf` предоставляет определенную информацию в диалоговых окнах установки.

5. Перетащите файл `demo.txt` в папку `User's Desktop` (Рабочий стол пользователя). Этот файл должен устанавливаться только после положительного ответа пользователя на вопрос о действительной необходимости выполнения установки, поэтому установите значение свойства `Condition` этого файла равным `CHECKBOXDEMO`. Условие `CHECKBOXDEMO` — то, которое может быть установлено пользователем. Значение должно быть записано прописными буквами. Файл устанавливается, только если условие `CHECKBOXDEMO` равно `true`. Позже мы определим диалоговое окно, в котором устанавливается это свойство.

6. Чтобы программа была доступна через меню Start⇨Programs, требуется ярлык для программы `MDI Editor`.

   В папке `Application Folder` выберите элемент Primary Output from MDI Editor (Основной вывод MDI Editor) и откройте меню Action⇨Create Shortcut to Primary output from MDI Editor (Действие⇨Создать ярлык для основного вывода MDI Editor). Установите `Wrox MDI Editor` в качестве значения свойства `Name` созданного ярлыка и перетащите этот ярлык в меню User's Programs (Программы пользователя).

Редактор File Types Editor

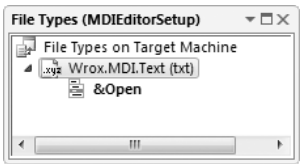


***Рис. 17.28**. Редактор File Types Editor с добавленным нестандартным расширением*

Если приложение использует нестандартные типы файлов и нужно зарегистрировать эти расширения для файлов, которые должны запускать приложение при двойном щелчке на них, можно использовать редактор File Types Editor (Редактор типов файлов), выбирая опцию меню View⇨Editor⇨File Types (Вид⇨Редактор⇨Типы файлов). Редактор File Types Editor с добавленным нестандартным расширением показан на рис. 17.28.

Редактор File Types Editor позволяет конфигурировать расширение файла, которое должно обрабатываться в приложении. Свойства расширений файлов описаны в табл. 17.2.

Таблица 17.2. Свойства расширений файлов

Свойство	Описание
`Name`	Добавляет полезное имя, описывающее тип файла. Это имя отображается в окне редактора File Types Editor и записывается в системный реестр. Следует выбирать уникальное имя. Примером для типов файлов `.doc` может служить `Word.Document.12`. Использование идентификатора программы (`ProgID`), как в примере файла Word, не обязательно. Можно применять также простой текст наподобие `wordhtmlfile`, который используется для расширения файла `.dochtml`
`Command`	Свойство `Command` позволяет указывать исполняемый файл, который должен запускаться при открытии пользователем файла данного типа
`Description`	Добавляет описание
`Extensions`	Это свойство определяет расширение файла, в котором приложение должно быть зарегистрировано. Расширение файла будет зарегистрировано в соответствующем разделе системного реестра
`Icon`	Указывает значок, который должен отображаться для данного расширения файла

Создание действий

После создания типов файлов в редакторе File Types Editor можно добавить необходимые действия. Действие, автоматически добавляемое по умолчанию — `Open` (Открыть). Однако можно добавить дополнительные действия, такие как `New` (Создать) и `Print` (Печать), или любые другие, подходящие для данной программы. Вместе с действиями необходимо определить свойства `Arguments` и `Verb`. Свойство `Arguments` указывает аргументы, передаваемые программе, которая зарегистрирована для данного расширения файла. Например, `%1` означает, что имя файла передается приложению. Свойство `Verb` указывает действие, которое должно быть выполнено. С действием печати может быть передано свойство `/print`, если приложение его поддерживает.

Следующее практическое занятие посвящено добавлению действия в программу установки `MDIEditor`. Нам необходимо зарегистрировать расширение файла, чтобы программу `MDIEditor` можно было использовать из проводника Windows для открытия файлов с расширением `.txt`. После выполнения такой регистрации на этих файлах можно будет дважды щелкнуть для их открытия, и приложение `MDIEditor` запустится автоматически.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Установка расширения файла

1. Используя пункт меню View⇨Editor⇨File Types, запустите редактор File Types Editor. Посредством меню Action⇨Add File Type (Действие⇨Добавить тип файла) добавьте новый тип файла, установив свойства, как показано в следующей таблице.

Свойство	Значение
(Name)	Wrox.MDIEditor.Text
Command	Primary output from MDIEditor
Description	Text Documents
Extensions	Txt

Можно также установить свойство `Icon`, чтобы определить значок для открытия файла, и тип MIME. Оставьте заданные по умолчанию значения свойств действия `Open`, чтобы имя файла передавалось в качестве аргумента приложения.

Редактор Launch Condition Editor

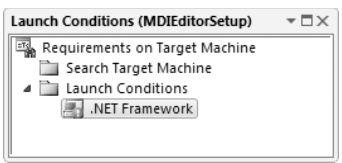


Рис. 17.29. *Окно редактора Launch Condition Editor*

Редактор Launch Condition Editor позволяет указывать требования к целевой системе, которые должны быть удовлетворены, прежде чем установка сможет быть выполнена. Запустите редактор Launch Condition Editor (рис. 17.29), выбирая пункт меню View⇨Editor⇨Launch Conditions (Вид⇨Редактор⇨Условия запуска).

Редактор предоставляет два раздела для указания требований: Search Target Machine (Искать на целевом компьютере) и Launch Conditions (Условия запуска). В первом разделе можно указывать конкретный файл или запись реестра, которые нужно искать. Во втором разделе определяется сообщение об ошибке, отображаемое в случае неудачи поиска. Ниже приведены некоторые условия запуска, которые можно определять посредством меню Action.

- File Launch Condition (Условие запуска файла). Выполняет в целевой системе поиск указанного файла перед запуском установки.

- Registry Launch Condition (Условие запуска реестра). Позволяет перед запуском установки потребовать проверку записей реестра.
- Windows Installer Launch Condition (Условие запуска программы установки Windows). Позволяет выполнить поиск компонентов программы установки Windows, которые обязательно должны присутствовать.
- .NET Framework Launch Condition (Условие запуска .NET Framework). Проверяет наличие установленной платформы .NET Framework в целевой системе.
- Internet Information Services Launch Condition (Условие запуска IIS). Проверяет наличие установленных информационных служб Интернета. Добавление этого условия запуска добавляет поиск конкретной записи реестра, определенной при установке информационных служб Интернета, а также условие проверки конкретной версии.

По умолчанию условие .NET Framework Launch Condition установлено, а его свойствам присвоены заранее определенные значения: `[VSDNETMSG]`, являющееся заранее определенным сообщением об ошибке, установлено в качестве значения свойства `Message`. Если платформа .NET Framework 4 не установлена, отображается сообщение, информирующее пользователя о необходимости установки .NET Framework. По умолчанию свойство `InstallUrl` установлено в `http://go.microsoft.com/fwlink/?LinkId=131000`, что позволяет пользователям без труда запустить установку .NET Framework.

Редактор User Interface Editor

Редактор User Interface Editor позволяет определять диалоговые окна, отображаемые при конфигурировании установки. В них можно информировать пользователей об условиях лицензионного соглашения и запрашивать пути установки и другую информацию для конфигурирования приложения.

В следующем практическом занятии мы запустим редактор User Interface Editor, используемый для конфигурирования диалоговых окон, которые открываются во время установки приложения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Запуск редактора User Interface Editor

1. Запустите редактор User Interface Editor, выбрав пункт меню View⇨Editor⇨User Interface (Вид⇨Редактор⇨Интерфейс пользователя).
2. С помощью редактора User Interface Editor установите свойства предопределенных диалоговых окон. Автоматически сгенерированные диалоговые окна и два доступных режима установки показаны на рис. 17.30.

Описание работы

Как показано на рис. 17.30, в данном случае доступны два режима установки: Install (Установка) и Administrative Install (Административная установка). Как правило, режим Install используется для установки приложения в целевой системе. Режим Administrative Install позволяет устанавливать образ приложения в разделяемой сетевой папке. После этого пользователи могут устанавливать приложение из сети.

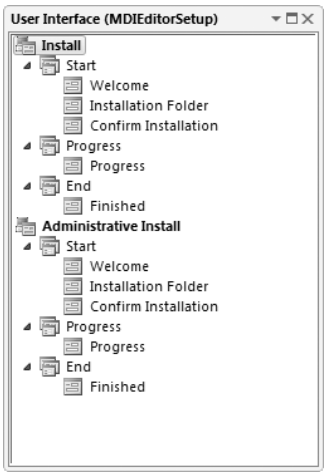


***Рис. 17.30**. Автоматически сгенерированные диалоговые окна и два доступных режима установки*

Оба режима установки состоят из трех этапов, на которых могут отображаться диалоговые окна: Start (Начало), Progress (Ход работ) и End (Завершение). Рассмотрим диалоговые окна, определенные по умолчанию.

- Диалоговое окно Welcome (Приветствие) отображает пользователю приветственное сообщение.

  Заданный по умолчанию текст приветствия можно заменить собственным сообщением. Пользователи могут только отменить установку или щелкнуть на кнопке Next (Далее).

- Во втором диалоговом окне, Installation Folder (Папка для установки), пользователи могут выбрать папку, в которой должно быть установлено приложение. При добавлении нестандартных диалоговых окон (их мы рассмотрим ниже) они должны быть добавлены перед этим диалоговым окном.

- Диалоговое окно Confirm Installation (Подтверждение установки) — последнее, отображаемое перед запуском установки.

- Диалоговое окно Progress (Ход работ) отображает элемент управления ходом работ, позволяющий пользователям наблюдать за протеканием установки.

- По завершении установки открывается диалоговое окно Finished (Установка завершена).

Диалоговые окна, определенные по умолчанию, открываются автоматически во время установки, даже если редактор User Interface Editor ни разу не открывался, но эти диалоговые окна должны быть сконфигурированы для отображения полезных сообщений, соответствующих приложению.

В следующем практическом занятии мы выполним конфигурирование диалоговых окон, которые открываются при установке приложения. В данном случае мы опустим ветвь Administrative Install и выполним только конфигурирование ветви типичной установки.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Конфигурирование диалоговых окон, определенных по умолчанию

1. Выберите диалоговое окно Welcome. Окно Properties содержит три свойства этого диалогового окна: `BannerBitmap`, `CopyrightWarning` и `WelcomeText`. В свойстве `BannerBitmap` щелкните на опции Browse (Обзор) поля со списком и выберите файл `wroxsetuplogo.bmp` из папки `Application Folder\Setup`. Хранящееся в этом файле растровое изображение будет отображаться в верхней части данного диалогового окна.

   Заданный по умолчанию текст свойства `CopyrightWarning` выглядит следующим образом:

   ```
   WARNING: This computer program is protected by copyright law and international
   treaties. Unauthorized duplication or distribution of this program, or any
   portion of it, may result in severe civil or criminal penalties, and will be
   prosecuted to the maximum extent possible under the law.
   ```

   ПРЕДУПРЕЖДЕНИЕ. Эта компьютерная программа защищена законами о защите авторских прав и международными соглашениями. Несанкционированное копирование или распространение этой программы или любой ее части может повлечь суровое административное или криминальное наказание, и будет подвергаться максимальному преследованию в рамках действующего законодательства.

   Этот текст отображается также в диалоговом окне Welcome. Если хотите отобразить более строгое предупреждение, измените текст. Свойство `WelcomeText` определяет

дополнительный текст, отображаемый в диалоговом окне. Его значение, определенное по умолчанию, выглядит следующим образом:

```
The installer will guide you through the steps required to install
[ProductName] on your computer.
```

Программа установки поможет выполнить действия, необходимые для установки [ProductName] на вашем компьютере.

Этот текст также можно изменить. Строка `[ProductName]` будет автоматически замещена свойством `ProductName`, которое было определено в свойствах проекта.

2. Выберите диалоговое окно Installation Folder. Это диалоговое окно имеет два свойства: `BannerBitmap` и `InstallAllUsersVisible`. Значение второго свойства, используемое по умолчанию — `true`. Если его установить равным `false`, установка приложения будет возможной только для пользователя, который зарегистрирован в системе во время выполнения установки. Укажите в качестве значения свойства `BannerBitmap` файл `wroxsetuplogo.bmp`, как это было выполнено для диалогового окна Welcome. Поскольку это свойство позволяет отображать растровое изображение в каждом диалоговом окне, измените свойство `BannerBitmap` и для всех остальных диалоговых окон.

Дополнительные диалоговые окна

Если вы разработали нестандартное диалоговое окно, программа установки Visual Studio Installer позволяет добавить его в процесс установки. Вообще говоря, для этого требуется более сложный инструмент, такой как InstallShield или Wise for Windows, но программа установки Visual Studio Installer позволяет добавлять и настраивать многие заранее определенные диалоговые окна посредством использования экрана Add Dialog (Добавление диалогового окна).

Выбор опции Start sequence (Последовательность запуска) в редакторе User Interface Editor и выбор пункта меню Action⇨Add Dialog (Действие⇨Добавить диалоговое окно) ведет к отображению диалогового окна Add Dialog (Добавление диалогового окна), показанного на рис. 17.31. Все эти диалоговые окна являются настраиваемыми.

Доступны диалоговые окна, содержащие два, три или четыре переключателя, диалоговые окна с флажками, отображающие до четырех флажков, и диалоговые окна с текстовыми полями, отображающие до четырех текстовых полей. Эти диалоговые окна можно конфигурировать, устанавливая их свойства.

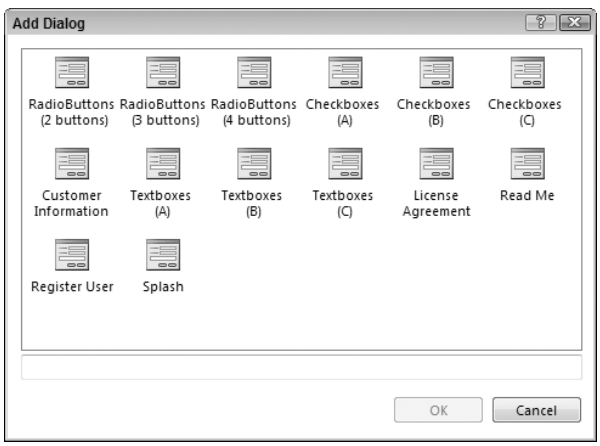


***Рис. 17.31**. Диалоговое окно Add Dialog*

Ниже кратко рассмотрены некоторые из диалоговых окон.

- Диалоговое окно Customer Information (Информация о клиенте) запрашивает пользователей об их имени и названии организации, а также серийный номер продукта. Если серийный номер продукта не предоставляется, текстовое поле Serial Number (Серийный номер) можно скрыть, устанавливая значение свойства `ShowSerialNumber` равным `false`.
- Диалоговое окно License Agreement (Лицензионное соглашение) позволяет пользователям принять условия лицензионного соглашения до начала процесса установки. Файл лицензионного соглашения определяется свойством `LicenseFile`.
- В диалоговом окне Register User (Регистрация пользователя) пользователи могут щелкнуть на кнопке Register Now (Зарегистрировать), чтобы запустить программу, которая определена в свойстве `Executable`. Пользовательская программа может пересылать данные на сервер FTP или передавать данные по электронной почте.
- Диалоговое окно Splash (Заставка) отображает экран заставки перед началом установки, используя при этом растровое изображение, указанное в свойстве `SplashBitmap`.

В следующем практическом занятии мы добавим несколько дополнительных диалоговых окон, таких как Read Me (Важная информация), License Agreement (Лицензионное соглашение) и Checkboxes (Флажки).

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Добавление других диалоговых окон

1. Выбирая команду меню Action⇨Add Dialog, добавьте в процесс запуска диалоговые окна Read Me, License Agreement и Checkboxes (A). С помощью перетаскивания определите последовательность открытия диалоговых окон в процессе запуска:

   `Welcome — Read Me — License Agreement — Checkboxes (A) — Installation Folder — Confirm Installation.`

2. Для всех этих диалоговых окон сконфигурируйте свойство `BannerBitmap`, как это делалось ранее. В качестве значения свойства `ReadmeFile` диалогового окна Read Me установите `readme.rtf` — файл, который ранее был добавлен в каталог `Application Folder\Setup`.
3. В качестве значения свойства `LicenseFile` диалогового окна License Agreement установите `license.rtf`.
4. Используйте диалоговое окно Checkboxes (A) для запроса пользователей о необходимости установки файла `demo.wroxtext` (который был помещен в папку `Desktop` пользователя). Измените свойства этого диалогового окна в соответствии со следующей таблицей:

Свойство	Значения
`BannerText`	`Optional Files`
`BodyText`	`Installation of optional files`
`Checkbox1Label`	`Do you want a demo file put onto the desktop?`
`Checkbox1Property`	`CHECKBOXDEMO`
`Checkbox2Visible`	`False`
`Checkbox3Visible`	`False`
`Checkbox4Visible`	`False`

Значение свойства Checkbox1Property устанавливается таким же, как и свойства Condition файла demo.wroxtext — это значение свойства Condition было установлено ранее при добавлении файла в пакет с помощью редактора File System Editor. Если пользователь установит этот флажок, значение свойства CHECKBOXDEMO будет равным true, и файл будет установлен. В противном случае значение будет равным false, и файл устанавливаться не будет.

Значение свойства CheckboxXVisible других флажков устанавливается равным false, поскольку нам нужен только один флажок.

Компоновка проекта

Теперь можно выполнить следующее практическое занятие, чтобы приступить к компоновке проекта программы установки.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Компоновка проекта**

1. Чтобы создать пакет для программы установки Windows, щелкните правой кнопкой мыши на проекте SimpleEditorSetup и выберите в контекстном меню пункт Build (Скомпоновать).
2. В случае успешной компоновки файлы setup.exe и WroxSimpleEditor.msi, а также readme.txt появятся в каталоге Debug (Отладка) или Release (Публикация) (в зависимости от настроек компоновки).

Описание работы

Исполняемый файл Setup.exe запускает установку MSI-файла базы данных WroxSimpleEditor.msi. Все файлы, добавленные в проект программы установки (за одним исключением), объединены и сжаты в файл MSI, поскольку в свойствах проекта было указано паковать файлы в файл установки (Package Files in Setup File). Единственное исключение — файл readme.txt, свойство PackageAs которого было изменено, чтобы его можно было прочесть непосредственно перед установкой приложения. Пакет установки платформы .NET Framework можно найти в подкаталоге DotNetFx.

Установка

Теперь можно приступить к установке приложения MDI Editor. Дважды щелкните на файле Setup.exe или выберите файл WroxMDIEditor.msi. Щелкните правой кнопкой мыши, чтобы открыть контекстное меню и выберите из него пункт Install (Установка). Установку можно запустить также из среды Visual Studio 2010, щелкая правой кнопкой на открытом проекте установки в окне Solution Explorer и выбирая из контекстного меню пункт Install.

Как будет показано в последующих разделах, все диалоговые окна содержат логотип Wrox, а вставленные диалоговые окна Read Me и License Agreement открываются с определенными при их конфигурировании файлами.

Окно Welcome

Первое открывающееся диалоговое окно — Welcome (Приветствие), показанное на рис. 17.32. При этом отображается логотип Wrox, который был вставлен установкой свойства BannerBitmap в соответствующее значение. Отображаемый текст определен свойствами WelcomeText и CopyrightWarning. Заголовок этого диалогового окна определяется свойством ProductName, установленным в свойствах проекта.

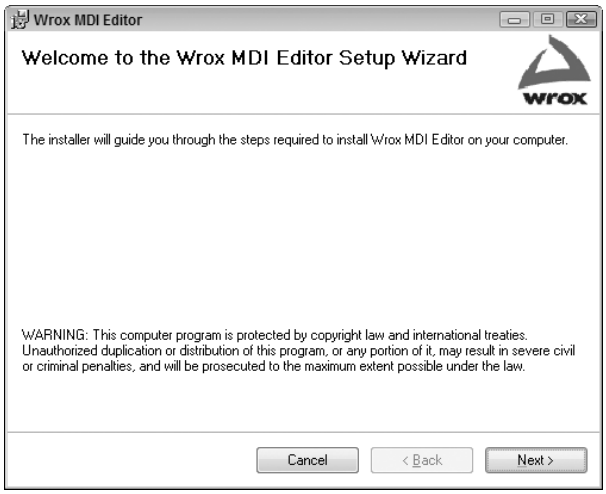


***Рис. 17.32**. Диалоговое окно* Welcome

Окно Read Me

После щелчка на кнопке Next (Далее) открывается диалоговое окно Read Me (рис. 17.33). Оно отображает содержимое файла `readme.rtf`, который был сконфигурирован посредством установки свойства `ReadmeFile`.

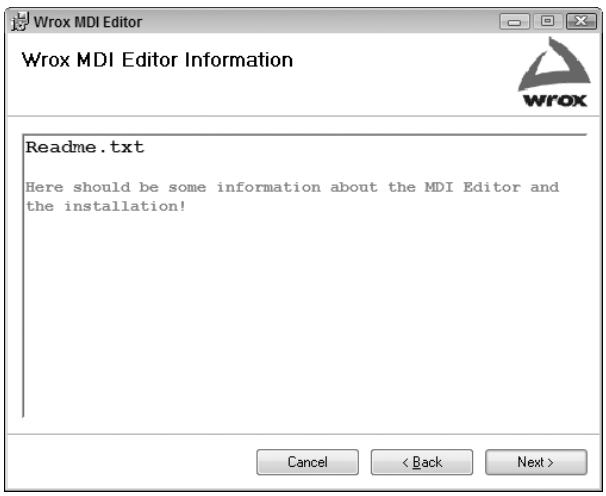


***Рис. 17.33**. Диалоговое окно* Read Me

Окно License Agreement

Третье открывающееся диалоговое окно — окно лицензионного соглашения License Agreement. Для него конфигурируются только свойства `BannerBitmap` и `LicenseFile`. Переключатели принятия условий лицензионного соглашения добавляются автоматически. Как видно на рис. 17.34, кнопка Next (Далее) остается отключенной до тех пор, пока не будет выбран переключатель I Agree (Я принимаю условия). Это функционирование диалогового окна определено автоматически.

***Рис. 17.34**. Диалоговое окно* License Agreement

Окно Optional Files

Принятие условий лицензионного соглашения и щелчок на кнопке Next (Далее) ведет к отображению диалогового окна Optional Files (Необязательные файлы), которое показано на рис. 17.35. При этом должен отобразиться текст, который был определен свойствами `BannerText`, `BodyText` и `Checkbox1Label`. Остальные флажки невидимы, поскольку значения соответствующих свойств `CheckboxVisible` установлены в `false`.

Установка флажка приведет к установке файла `demo.txt` на рабочий стол.

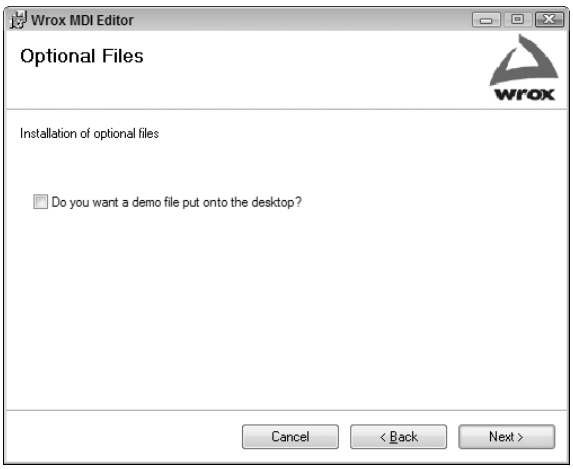


***Рис. 17.35**. Диалоговое окно* Optional Files

Окно Select Installation Folder

В диалоговом окне Select Installation Folder (Выберите папку для установки), показанном на рис. 17.36, пользователи могут выбрать путь для установки приложения. Для этого диалогового окна можно настраивать только свойство `BannerBitmap`. Путь для установки, отображаемый по умолчанию — `[Program Files]\[`*`Изготовитель`*`]\[`*`Имя продукта`*`]`.

Пользователи могут также указывать, должна установка приложения выполняться для любого пользователя или же только для зарегистрированного пользователя. В зависимости от ответа на этот вопрос ярлык программы будет помещен в персональный каталог пользователя или в каталог `All Users` (Все пользователи).

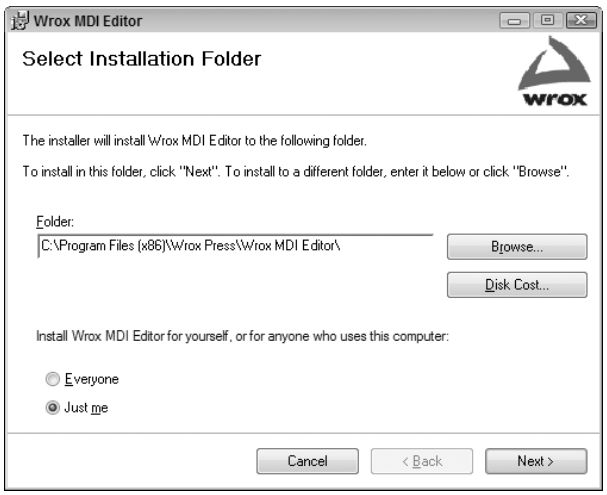


Рис. 17.36. *Диалоговое окно* Select Installation Folder

Окно Disk Cost

Щелчок на кнопке Disk Cost (Объем, занимаемый на диске) ведет к открытию диалогового окна Disk Cost, показанного на рис. 17.37. В нем отображается дисковое пространство всех жестких дисков и вычисляется дисковое пространство, необходимое для установки компонентов приложения на каждом из дисков. Это облегчает пользователю выбор диска, на котором должно быть установлено приложение.

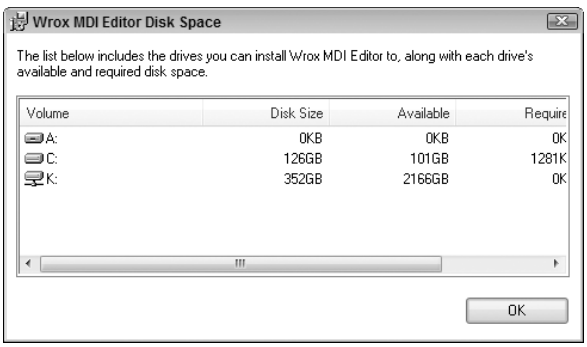


Рис. 17.37. *Диалоговое окно* Disk Cost

Окно Confirm Installation

Диалоговое окно Confirm Installation (Подтверждение установки), показанное на рис. 17.38 — это последнее окно, которое открывается перед началом установки. Оно не содержит никаких дополнительных вопросов и предоставляет последнюю возможность отменить установку до ее начала.

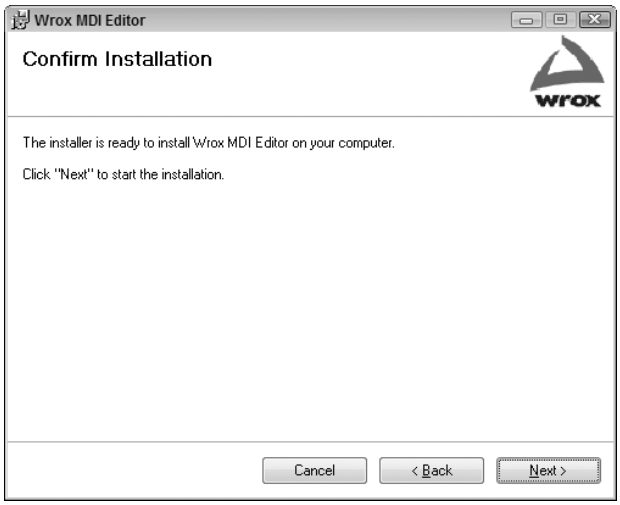


***Рис. 17.38**. Диалоговое окно* Confirm Installation

Окно Progress

Диалоговое окно Installing (Выполняется установка), показанное на рис. 17.39, отображает элемент управления ходом работ во время установки, показывая пользователю, что установка продолжается, и давая приближенное представление о ее продолжительности. Редактор `MDI Editor` — небольшая программа, поэтому данное диалоговое окно завершит свою работу очень быстро.

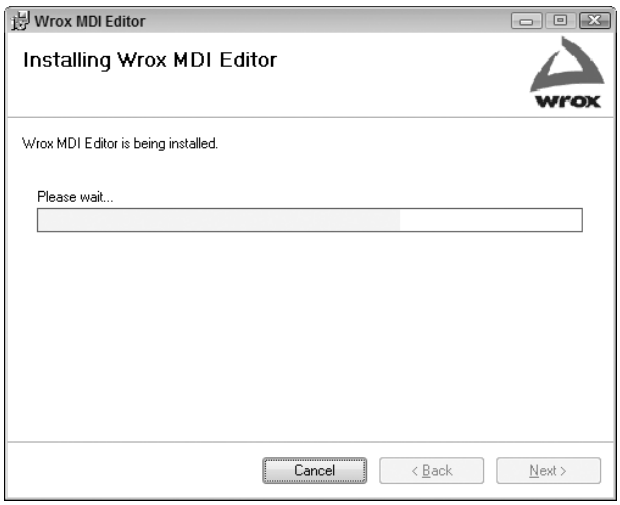


***Рис. 17.39**. Диалоговое окно* Installing

Окно Installation Complete

После успешной установки откроется последнее диалоговое окно — Installation Complete (Установка завершена), — показанное рис. 17.40.

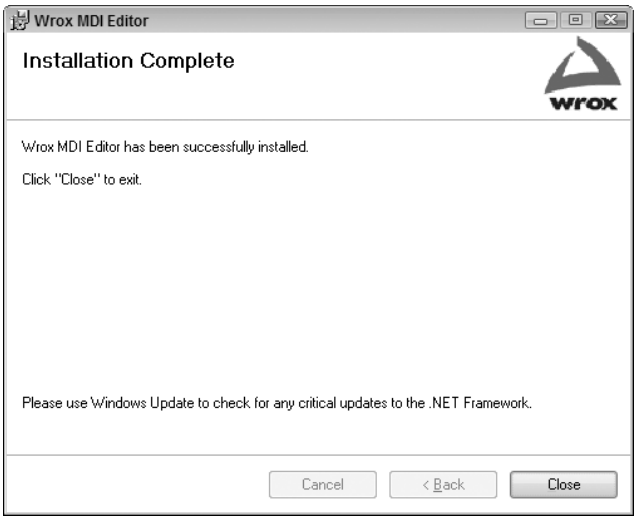


Рис. 17.40. *Диалоговое окно* Installation Complete

Выполнение приложения

Редактор можно запустить, выбрав пункт меню Start⇨All Programs⇨Wrox MDI Editor (Пуск⇨Все программы⇨MDI-редактор Wrox). Поскольку мы зарегистрировали расширение файла, существует и другой способ запуска приложения: двойной щелчок на файле с расширением `.txt`. Чтобы выбрать приложение, которое должно быть использовано для открытия файла в данном случае, выберите файл в проводнике Windows и откройте контекстное меню. Выберите пункт контекстного меню Open with... (Открыть с помощью), а затем нужное приложение. Чтобы определить приложение, которое должно открывать файл при двойном щелчке на нем, откройте контекстное меню, выберите его пункт Open with..., а затем пункт Choose default program... (Выбрать программу по умолчанию).

Если в диалоговом окне Optional Files (Необязательные файлы) соответствующий флажок был установлен, файл `demo.txt` можно будет найти на рабочем столе.

Удаление приложения

Если хотите избавиться от приложения `MDI Editor`, можно воспользоваться значком Add/Remove Programs панели управления. Выберите `MDI Editor` и щелкните на кнопке Remove (Удалить).

Резюме

В этой главе было описано использование технологии развертывания ClickOnce и функционирование программы установки Windows, включая создание пакетов программы установки с помощью Visual Studio 2010. Программа установки Windows позволяет легко выполнять стандартизованные установки, отмены и восстановления приложений.

ClickOnce — новая технология, которая облегчает установку Windows-приложений без необходимости регистрации в качестве системного администратора. Эта технология позволяет без труда осуществлять как развертывание, так и обновление клиентских приложений.

Если требуется больше функциональных средств, чем обеспечивает технология ClickOnce, в этом поможет программа установки Windows. Программа установки Visual Studio 2010 ограничена в своей функциональности и не обладает всеми возможностями

программы установки Windows, но для многих приложений ее функций вполне достаточно. Несколько редакторов позволяют конфигурировать сгенерированный файл для программы установки Windows. Редактор File System Editor позволяет указывать все файлы и ярлыки. С помощью редактора Launch Conditions Editor можно определять некоторые обязательные компоненты. Редактор File Types Editor служит для регистрации расширений файлов для различных приложений. Редактор User Interface Editor позволяет легко адаптировать диалоговые окна, используемые в процессе установки.

Упражнения

1. В чем преимущества развертывания по технологии ClickOnce?
2. Что определяется манифестом ClickOnce?
3. Когда необходимо применять программу установки Windows?
4. Какие различные редакторы можно использовать для создания пакета программы установки Windows с помощью Visual Studio?

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Технология ClickOnce	Технология ClickOnce может использоваться для развертывания приложений без применения административных прав доступа. Установка приложения Windows Forms или WPF выполняется простым щелчком на ссылке на веб-странице. Это — огромное преимущество технологии ClickOnce, поскольку развертывание приложений перестает быть кошмаром администраторов IT-служб. Развертывание с помощью ClickOnce может быть создано в разделе Publish окна свойств проекта.
Пакет программы установки Windows	Программа установки Windows позволяет устанавливать в системе приложения совместного использования. Эта технология позволяет устанавливать компоненты приложений, которые требуют административных привилегий. Пакет программы установки можно легко создать с помощью шаблона Setup Project (Проект установки) программы установки Visual Studio.
Настройка диалоговых окон установки	Шаблон Setup Project среды Visual Studio предоставляет ряд предварительно определенных диалоговых окон для установки, которые могут быть настроены с помощью таких свойств, как текст заявления об авторских правах и логотипы, отображаемые во время процесса установки.

ЧАСТЬ III

Программирование веб-приложений

В ЭТОЙ ЧАСТИ...

18

Программирование веб-приложений с использованием технологии ASP.NET

В ЭТОЙ ГЛАВЕ...

- Обзор разработки с использованием технологии ASP.NET
- Использование элементов управления сервером ASP.NET
- Обратная отправка ASP.NET различным страницам
- Создание обратных Ajax-отправок ASP.NET
- Проверка достоверности пользовательского ввода
- Управление состоянием
- Добавление стилей к веб-странице
- Использование мастер-страниц
- Реализация навигации по страницам
- Аутентификация и авторизация пользователей
- Запись и чтение баз данных SQL Server

Windows Forms — это технология создания Windows-приложений, а технология ASP.NET позволяет создавать веб-приложения, которые отображаются в любом браузере. Технология ASP.NET позволяет создавать веб-приложения подобно тому, как разрабатываются Windows-приложения. Это становится возможным благодаря серверным элементам управления, которые абстрагируются от HTML-разметки и имитируют поведение элементов управления Windows. Конечно, между Windows- и веб-приложениями все же существует много различий, которые обусловлены лежащими в основе веб-приложений технологиями — HTTP и HTML.

Эта глава содержит обзор программирования веб-приложений с применением технологии ASP.NET. В ней описано использование веб-элементов управления, обработка средств управления состоянием (которое значительно отличается от обработки в Windows-приложениях), выполнение аутентификации и считывание и запись информации в базах данных.

Обзор веб-приложений

Веб-приложение вынуждает веб-сервер отправлять HTML-разметку клиенту. Эта разметка отображается в веб-браузере, таком как Internet Explorer. Когда в браузере пользователь вводит строку URL-адреса, HTTP-запрос отравляется веб-серверу. HTTP-запрос содержит имя запрошенного файла и такую дополнительную информацию, как строка идентификации клиентского приложения, языки, поддерживаемые клиентом, и дополнительные данные, относящиеся к запросу. Веб-сервер возвращает HTTP-ответ, который содержит HTML-разметку, интерпретируемую веб-браузером для отображения текстовых полей, кнопок и списков.

ASP.NET — это технология динамического создания веб-страниц с помощью кода серверной стороны. Разработка этих веб-страниц может выполняться во многом подобно клиентским программам Windows. Вместо того чтобы непосредственно иметь дело с HTTP-запросом и ответом и вручную создавать HTML-разметку, предназначенную для отправки клиенту, можно использовать такие элементы управления, как `TextBox`, `Label`, `ComboBox` и `Calendar`, которые создают HTML-разметку автоматически.

Исполняющая среда ASP.NET

Чтобы технологию ASP.NET можно было применять к веб-приложениям на клиентской системе, в ней должен присутствовать только простой веб-браузер. Можно использовать Internet Explorer, Opera, Netscape Navigator, Firefox или любой другой веб-браузер, который поддерживает HTML. Клиентская система не требует установленной среды .NET.

В то же время серверная система нуждается в исполняющей среде ASP.NET. Если в системе установлены информационные службы Интернета (IIS), исполняющая среда ASP.NET конфигурируется вместе с сервером во время установки платформы .NET Framework. Во время разработки не обязательно иметь дело с IIS, поскольку Visual Studio предоставляет собственный сервер разработки веб-приложений ASP.NET, который можно использовать для тестирования и отладки приложений.

Чтобы понять, как действует исполняющая среда ASP.NET, рассмотрим типичный веб-запрос из браузера (рис. 18.1). Клиент запрашивает с сервера файл, например, `default.aspx`. Обычно все веб-страницы ASP.NET имеют расширение файла `.aspx`. Поскольку это расширение файла зарегистрировано в IIS или известно серверу разработки веб-приложений ASP.NET, исполняющая среда и рабочие процессы ASP.NET оказываются в курсе происходящего. При первом запросе файла `default.aspx` синтаксический анализатор ASP.NET запускается, и компилятор компилирует запрошенный файл и файл C#, связанный с файлом `.aspx`, создавая сборку. Затем компилятор JIT исполняющей среды .NET компилирует сборку в собственный код. Сборка содержит класс `Page`, который вызывается

для возвращения HTML-разметки клиенту. После этого объект `Page` уничтожается. Однако сборка сохраняется для обработки последующих запросов, что устраняет необходимость в повторной ее компиляции.

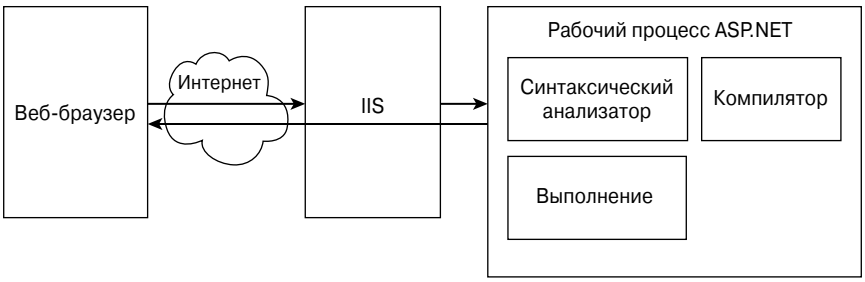


Рис. 18.1. *Типичный веб-запрос из браузера*

Создание простой страницы

В следующем практическом занятии будет создаваться простая веб-страница. В простом приложении, используемом в настоящей и следующей главе, будет создан простой веб-сайт Events, в котором участники могут регистрироваться при проведении различных мероприятий.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание простой веб-страницы

1. В среде Visual Studio создайте новый веб-проект, выбрав пункт меню File⇨New⇨Project (Файл⇨Создать⇨Проект). В диалоговом окне New Project (Новый проект), показанном на рис. 18.2, выберите категорию Visual C#, подкатегорию Web, а затем шаблон ASP.NET Empty Web Application (Пустое веб-приложение ASP.NET). Назначьте проекту имя `EventRegistrationWeb`.

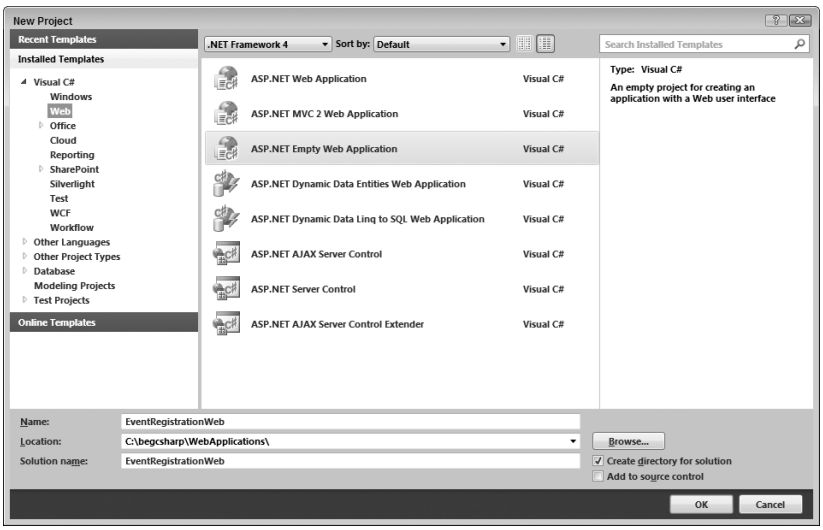


Рис. 18.2. *Создание нового проекта в диалоговом окне* New Project

2. После создания веб-проекта с помощью пункта меню Project⇨Add New Item (Проект⇨Добавить новый элемент) создайте новую веб-страницу, выберите шаблон Web Form (рис. 18.3) и назначьте ему имя `Registration.aspx`.

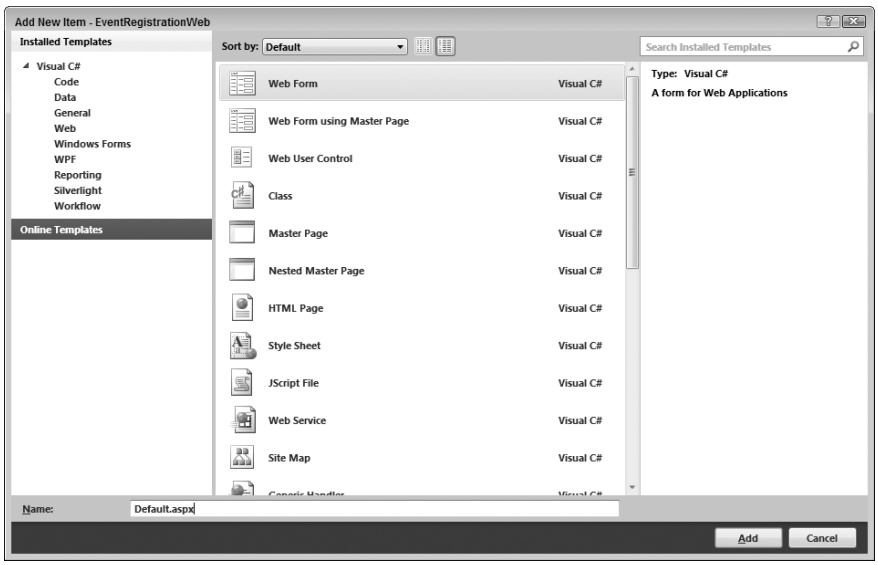


Рис. 18.3. *Добавление веб-формы*

3. Упорядочение элементов управления удобно осуществлять с помощью таблицы. Щелкните в представлении конструктора и добавьте таблицу, выбрав пункт меню Table⇨Insert Table (Таблица⇨Вставить таблицу). В диалоговом окне Insert Table (Вставка таблицы) определите пять строк и два столбца, как показано на рис. 18.4.

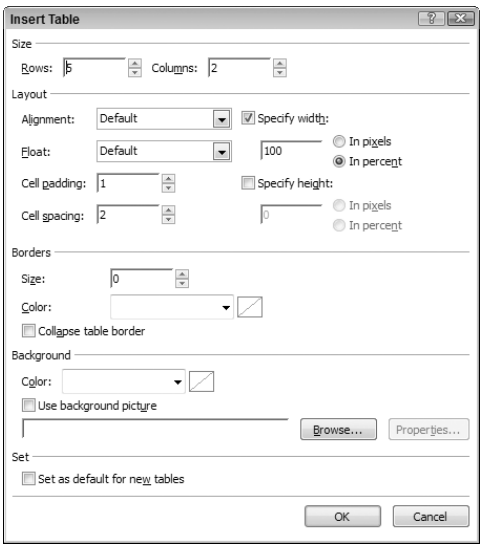


Рис. 18.4. *Добавление таблицы*

4. Добавьте в таблицу четыре элемента управления `Label`, три элемента управления `TextBox` и элементы управления `DropDownList` и `Button`, как показано на рис. 18.5.

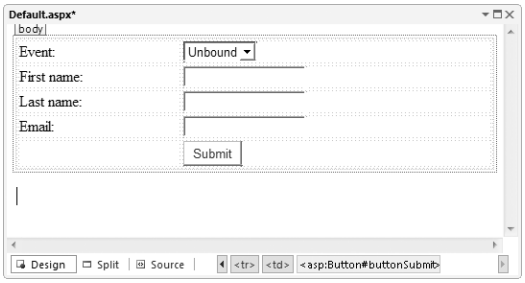


Рис. 18.5. *Компоновка веб-страницы*

5. Определите свойства элементов управления, как показано в следующей таблице:

Тип элемента управления	Свойство (ID)	Свойство Text
Label	labelEvent	Event:
Label	labelFirstName	First name:
Label	labelLastName	Last name:
Label	labelEmail	Email:
DropDownList	dropDownListEvents	
TextBox	textFirstName	
TextBox	textLastName	
TextBox	textEmail	
Button	buttonSubmit	Submit

6. В окне Properties (Свойства) элемента управления `DropDownList` выберите свойство `Items` и в диалоговом окне List-Item Collection Editor (Редактор коллекции элементов списка) введите строки `Introduction to ASP.NET`, `Introduction to Windows Azure` и `Take off to .NET 4.0`, как показано на рис. 18.6.

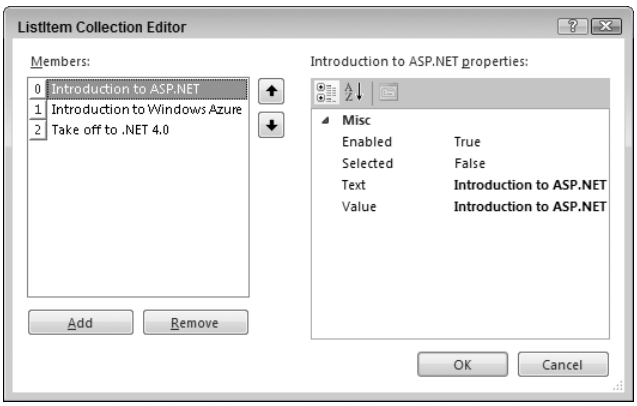


Рис. 18.6. *Ввод элементов списка*

7. Переключите редактор на отображение исходного кода и убедитесь, что сгенерированный код выглядит, как показано в следующем фрагменте.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Registration.aspx.cs"
        Inherits="EventRegistrationWeb.Registration" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
  <style type="text/css">
    .style1
    {
      width: 100%;
    }
  </style>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    <table class="style1">
      <tr>
        <td>
          <asp:Label ID="labelEvent" runat="server" Text="Event:">
          </asp:Label>
        </td>
        <td>
          <asp:DropDownList ID="dropDownListEvents" runat="server">
            <asp:ListItem>Introduction to ASP.NET</asp:ListItem>
            <asp:ListItem>Introduction to Windows Azure</asp:ListItem>
            <asp:ListItem>Take off to .NET 4.0</asp:ListItem>
          </asp:DropDownList>
        </td>
      </tr>
      <tr>
        <td>
          <asp:Label ID="labelFirstName" runat="server"
            Text="First name:"></asp:Label>
        </td>
        <td>
          <asp:TextBox ID="textFirstName" runat="server"></asp:TextBox>
        </td>
      </tr>
      <tr>
        <td>
          <asp:Label ID="labelLastName" runat="server" Text="Last name:">
          </asp:Label>
        </td>
        <td>
          <asp:TextBox ID="textLastName" runat="server"></asp:TextBox>
        </td>
      </tr>
      <tr>
        <td>
          <asp:Label ID="labelEmail" runat="server" Text="Email:">
          </asp:Label>
        </td>
        <td>
          <asp:TextBox ID="textEmail" runat="server"></asp:TextBox>
        </td>
      </tr>
```

```
        <tr>
          <td>
             </td>
          <td>
            <asp:Button ID="buttonSubmit" runat="server" Text="Submit" />
          </td>
        </tr>
      </table>
    </div>
    </form>
</body>
</html>
```

Фрагмент кода ***Registration.aspx***

8. Прежде чем запускать приложение, откройте окно свойств проекта и перейдите на вкладку настроек Web, показанную на рис. 18.7. Удостоверьтесь, что в группе Start action (Действие по запуску) выбран переключатель Current page (Текущая страница), а в группе Servers (Серверы) для конфигурирования указан сервер Visual Studio Development Server (Сервер разработки Visual Studio).

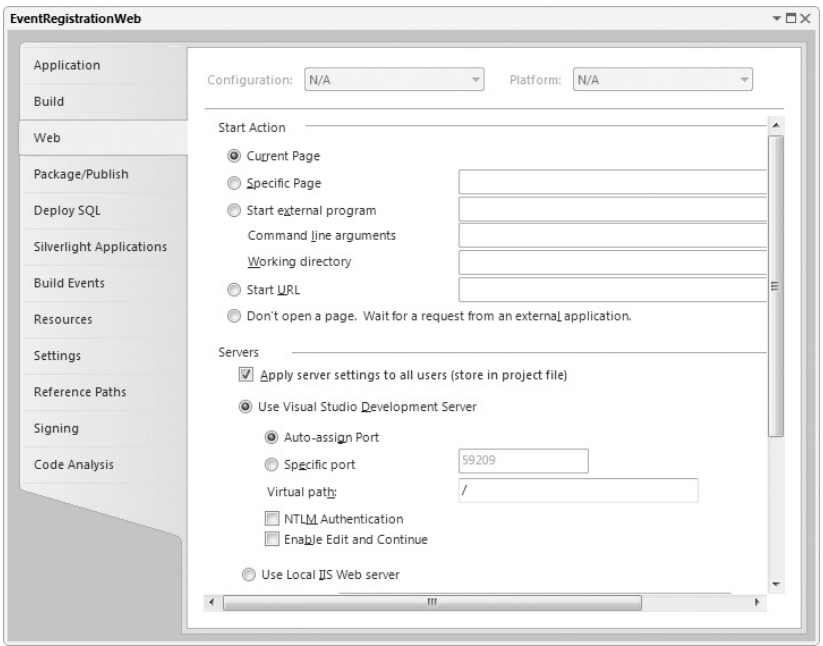


Рис. 18.7. *Вкладка* Web *окна свойств проекта*

9. Снова откройте в редакторе файл `Registration.aspx`. Запустите веб-приложение, выбрав пункт меню Debug⇨Start Without Debugging (Отладка⇨Запустить без отладки). При запуске приложения автоматически запускается веб-сервер разработки ASP.NET Development Server. Его значок отобразится в панели задач проводника Windows. Дважды щелкните на этом значке, чтобы открыть диалоговое окно, показанное на рис. 18.8. Оно отображает физический и виртуальный пути к веб-серверу, и прослушиваемый этим сервером порт. Это диалоговое окно можно использовать также для остановки веб-сервера.

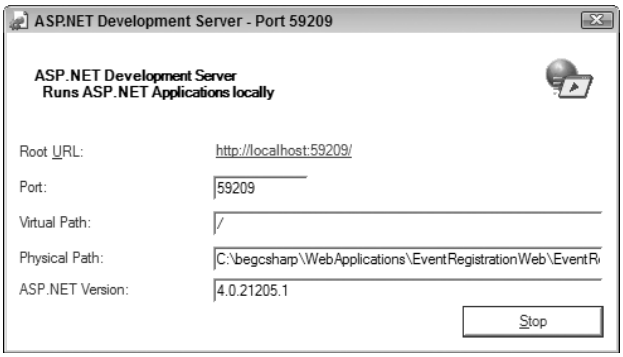


Рис. 18.8. *Запуск веб-сервера разработки*

В результате запуска приложения браузер Internet Explorer отобразит веб-страницу, как показано на рис. 18.9. HTML-разметку можно просмотреть, выбрав пункт меню View⇨Source (Вид⇨Исходный код). Вы увидите, что элементы управления серверной стороны были преобразованы в чистую HTML-разметку.

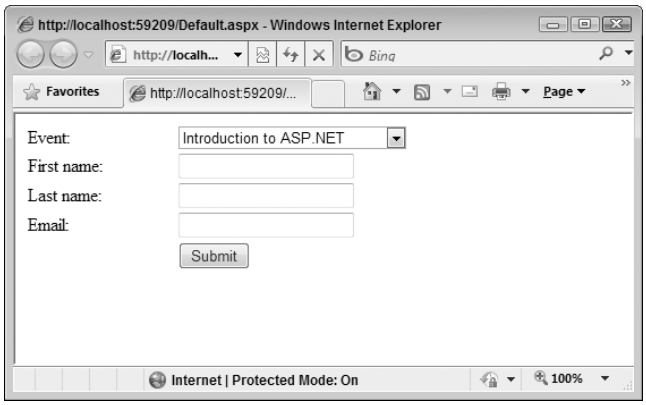


Рис. 18.9. *Готовая веб-страница*

Описание работы

Первой строкой файла `Registration.aspx` является директива `Page`:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Registration.aspx.cs"
        Inherits="EventRegistrationWeb.Registration" %>
```

Фрагмент кода ***Registration.aspx***

Эта директива определяет язык программирования и используемые классы. Как будет показано ниже, свойство `AutoEventWireup="true"` автоматически связывает обработчики событий с конкретными именами методов. `Inherits="EventRegistrationWeb.Registration"` означает, что класс, который динамически генерируется из файла ASPX, является производным от базового класса `Registration`. Этот базовый класс определен в файле отделенного кода `Registration.aspx.cs`, как указано в свойстве `CodeFile`. Позднее в этой главе мы добавим код обработчика в файл `.cs`. Сгенерированный файл отделенного кода `Registration.aspx.cs` имеет следующий вид.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace EventRegistrationWeb
{
  public partial class Registration : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
    }
  }
}
```

Фрагмент кода ***Registration.aspx.cs***

Ключевое слово `partial`*, использованное в приведенном коде, описано в главе 10.*

Код страницы ASPX имеет приведенный ниже вид. Клиент получает простую HTML-разметку в том виде, в каком он существует. При отправке страницы клиенту из дескриптора `<head>` лишь удаляется атрибут `runat="server"`.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
  <style type="text/css">
    .style1
    {
      width: 100%;
    }
  </style>
</head>
<body>
```

Фрагмент кода ***Registration.aspx***

В коде присутствуют и другие HTML-элементы с атрибутом `runat="server"`, такие как `<form>`. С помощью этого атрибута элемент управления сервера ASP.NET связывается с HTML-дескриптором. Элемент управления можно использовать для написания кода серверной стороны. В основе элемента `<form>` лежит объект типа `System.Web.UI.HtmlControls.HtmlForm`. Этот объект имеет имя переменной `form1`, определяемое атрибутом `id`. Объект `form1` можно использовать для вызова методов и свойств класса `HtmlForm`.

Объект `HtmlForm` создает дескриптор `<form>`, отправляемый клиенту:

```
<form id="form1" runat="server">
```

Фрагмент кода ***Default.aspx***

Естественно, атрибут `runat` клиенту не отправляется.

Стандартные элементы управления, помещаемые из панели инструментов на поверхность проектирования визуального конструктора форм (Forms Designer), содержат элементы, начинающиеся с последовательности символов `<asp:` — `<asp:Label>` и `<asp:DropDownList>`. Это элементы управления веб-приложений ASP.NET, связанные с классами .NET пространства имен `System.Web.UI.WebControls`. Элемент управления `<asp:Label>` представлен классом `Label`, а элемент `<asp:DropDownList>` — классом `DropDownList`.

```
<td>
    <asp:Label ID="labelEvent" runat="server" Text="Event:"></asp:Label>
</td>
<td>
    <asp:DropDownList ID="dropDownListEvents" runat="server">
        <asp:ListItem>Introduction to ASP.NET</asp:ListItem>
        <asp:ListItem>Introduction to Windows Azure</asp:ListItem>
        <asp:ListItem>Take off to .NET 4.0</asp:ListItem>
    </asp:DropDownList>
</td>
```

Фрагмент кода ***Registration.aspx***

<asp:Label> не отправляет клиенту элемент <asp:Label>, поскольку он не является допустимым HTML-элементом. Вместо этого он возвращает дескриптор . Аналогично, <asp:DropDownList> возвращает элемент <select>, а <asp:TextBox> — элемент <input type="text">.

Пространства имен System.Web.UI.HtmlControls и System.Web.UI.WebControls в ASP.NET содержат классы пользовательского интерфейса. Оба эти пространства имен имеют ряд аналогичных элементов управления, называемых также серверными HTML-элементами управления и серверными веб-элементами управления. Примером серверного HTML-элемента управления может служить HtmlInputText, а серверного веб-элемента управления — TextBox. Серверные HTML-элементы управления предоставляют методы и свойства, аналогичные методам и свойствам элементов управления HTML. Набор серверных веб-элементов управления предоставляет значительно более сложные элементы управления, такие как Calendar, DataGrid и Wizard. Если элемент управления не нуждается в написании кода серверной стороны, а лишь в коде JavaScript, можно ограничиться элементами управления HTML и не добавлять в генерируемый код атрибут runat="server".

Серверные элементы управления

В табл. 18.1 приведены описания некоторых основных серверных веб-элементов управления, доступных в среде ASP.NET, и возвращаемую ими HTML-разметку.

Таблица 18.1. Основные серверные веб-элементы управления

Элемент управления	HTML-разметка	Описание
Label		Возвращает элемент span, содержащий текст
Literal	static text	Возвращает простой статический текст. Этот элемент управления позволяет преобразовывать содержимое в соответствии с клиентским приложением
TextBox	<input type="text">	Возвращает HTML-элемент <input type="text">, посредством которого пользователь может вводить определенные значения. Можно создать обработчик события изменения текста
Button	<input type="submit">	Отправляет серверу значения формы
LinkButton	<a href="javascript: __dopostback()">	Создает дескриптор привязки, который включает в себя код JavaScript для выполнения обратной отправки на сервер

Окончание табл. 18.1

Элемент управления	HTML-разметка	Описание
ImageButton	<input type="image">	Генерирует дескриптор input типа image для отображения изображения, на которое осуществлена ссылка
HyperLink	<a>	Создает простой дескриптор привязки, ссылающийся на веб-страницу
DropDownList	<select>	Создает дескриптор select, посредством которого пользователь видит один элемент и может выбирать один элемент из нескольких, щелкая на раскрывающемся списке
ListBox	<select size="">	Создает дескриптор select с атрибутом size, который одновременно отображает несколько элементов
CheckBox	<input type="checkbox">	Возвращает элемент input типа checkbox для отображения кнопки, которая может быть выбрана или не выбрана. Вместо CheckBox можно использовать элемент управления CheckBoxList, который создает таблицу, состоящую из нескольких элементов CheckBox
RadioButton	<input type="radio">	Возвращает элемент input типа radio. Только один переключатель из группы может быть выбран. Аналогично CheckBoxList, элемент RadioButtonList предоставляет список кнопок
Image	<img src="">	Возвращает дескриптор img для отображения клиенту GIF- или JPG-файла
Calendar	<table>	Отображает полный календарь, в котором можно выбирать дату, изменять отображаемый месяц и т.п. Для вывода генерируется HTML-таблица с внедренным в нее кодом JavaScript
TreeView	<table>	Возвращает дескриптор div, который, в зависимости от своего содержимого, включает в себя несколько дескрипторов table. Код JavaScript используется для открытия и закрытия дерева элементов в клиенте

Обратная отправка ASP.NET

Серверные веб-элементы управления могут содержать обработчики событий, вызываемые на сервере. Элемент управления Button включает в себя событие Click, DropDownList предоставляет событие SelectedIndexChanged, а TextBox — событие TextChanged.

Событие генерируется на сервере только при обратной отправке. Когда значение в текстовом поле изменяется, событие `TextChanged` генерируется не немедленно, а только тогда, когда форма передается и отправляется серверу, что происходит при щелчке на кнопке Submit (Отправить). Прежде чем вызывать соответствующий обработчик события, исполняющая среда ASP.NET проверяет, что состояние элемента управления изменилось. Например, если состояние выбора элемента `DropDownList` изменилось, вызывается событие `SelectedIndexChanged`. Соответственно, событие `TextChanged` вызывается при изменении значения текстового поля.

Если желательно, чтобы событие изменения немедленно отправлялось серверу (например, при изменении выбора элемента `DropDownList`), свойство `AutoPostback` можно установить в `true`. В результате клиентская часть JavaScript используется для немедленной передачи данных формы на сервер. Конечно, при этом объем рабочего трафика возрастает, поэтому такую функциональность следует применять осмотрительно.

Для сравнения старых значений элемента управления с новыми после возвращения страницы серверу используется состояние представления. Состояние представления — это скрытое поле, которое отправляется браузеру вместе с содержимым страницы. При отправке страницы клиенту состояние представления содержит те же значения, что и элементы управления внутри формы. При обратной отправке серверу состояние представления отправляется серверу вместе с новыми значениями элементов управления. В результате можно проверить, изменились ли значения, и появляется возможность вызова обработчика события.

До сих пор пример приложения отправлял клиенту только простую страницу. Теперь придется иметь дело с результатом действий по вводу, выполненных пользователем. В первом примере ввод пользователя отображается в той же странице, а используется другая страница. В следующем практическом занятии мы отобразим ввод пользователя.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Отображение ввода пользователя

1. Откройте в Visual Studio ранее созданное веб-приложение `EventRegistrationWeb`.
2. Чтобы отобразить ввод пользователя для регистрации события, добавьте в веб-страницу `Registration.aspx` метку по имени `labelResult`. Очистите свойство `Text` этой метки.
3. Дважды щелкните на кнопке Submit (Отправить), чтобы добавить в эту кнопку обработчик события `Click`, и поместите в обработчик следующий код внутри файла `Registration.aspx.cs`:

```
protected void buttonSubmit_Click(object sender, EventArgs e)
{
   string selectedEvent = dropDownListEvents.SelectedValue;
   string firstName = textFirstName.Text;
   string lastName = textLastName.Text;
   string email = textEmail.Text;
   labelResult.Text = String.Format("{0} {1} selected the event {2}",
                      firstName, lastName, selectedEvent);
}
```

Фрагмент кода ***Registration.aspx.cs***

4. Снова запустите веб-страницу в Visual Studio. После ввода данных и щелчка на кнопке Submit эта же страница отобразит ввод пользователя в новой метке.

Описание работы

Двойной щелчок на кнопке Submit добавляет атрибут `OnClick` в элемент `<asp:Button>` в файле `Registration.aspx`:

```
<asp:Button ID="buttonSubmit" runat="server" Text="Submit"
      onclick="buttonSubmit_Click" />
```

Фрагмент кода ***Registration.aspx***

В серверном веб-элементе управления атрибут `OnClick` определяет событие `Click` серверной стороны, которое будет вызываться при щелчке на кнопке.

В реализации метода `buttonSubmit_Click()` значения элементов управления можно считывать посредством свойств. `dropDownListEvents` — переменная, которая ссылается на элемент управления `DropDownList`. В ASPX-файле идентификатор установлен в `dropDownListEvents`, поэтому переменная создается автоматически. Свойство `SelectedValue` возвращает текущий выбор. Свойство `Text` элементов управления `TextBox` возвращает строки, которые были введены пользователем.

```
string selectedEvent = dropDownListEvents.SelectedValue;
string firstName = textFirstName.Text;
string lastName = textLastName.Text;
string email = textEmail.Text;
```

Фрагмент кода ***Registration.aspx.cs***

Как и ранее, метка `labelResult` имеет свойство `Text`, в котором устанавливается результат:

```
labelResult.Text = String.Format("{0} {1} selected the event {2}",
      firstName, lastName, selectedEvent);
```

Фрагмент кода ***Registration.aspx.cs***

Вместо отображения результатов на той же странице ASP.NET позволяет легко отобразить результаты на другой странице, как показано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Отображение результатов на второй странице

1. Создайте новую форму `WebForm` с именем `ResultsPage.aspx`
2. Добавьте в эту форму метку по имени `labelResult`.
3. В метод `Page_Load` класса `ResultsPage` добавьте код, как показано в следующем фрагменте:

```
using System;
using System.Web.UI.WebControls;

namespace EventRegistrationWeb
{
  public partial class ResultsPage : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      try
      {
        DropDownList dropDownListEvents =
          (DropDownList)PreviousPage.FindControl("dropDownListEvents");
        string selectedEvent = dropDownListEvents.SelectedValue;
        string firstName = ((TextBox)PreviousPage.FindControl(
                           "textFirstName")).Text;
```

```
            string lastName = ((TextBox)PreviousPage.FindControl(
                               "textLastName")).Text;
            string email = ((TextBox)PreviousPage.FindControl(
                               "textEmail")).Text;
            labelResult.Text = String.Format("{0} {1} selected the event {2}",
                               firstName, lastName, selectedEvent);
          }
          catch
          {
            labelResult.Text = "The originating page must contain " +
                 "textFirstName, textLastName, textEmail controls";
          }
        }
      }
    }
```

Фрагмент кода ***ResultsPage.aspx.cs***

4. Установите свойство `PostbackUrl` кнопки Submit страницы `Registration.aspx` в `ResultsPage.aspx`.
5. Обработчик события `Click` можно удалить из кнопки Submit, поскольку он больше не нужен.
6. Запустите страницу `Default.aspx`, введите какие-то данные и щелкните на кнопке Submit. Произойдет переадресация к странице `ResultsPage.aspx`, где отображаются введенные данные.

Описание работы

В среде ASP.NET элемент управления `Button` реализует свойство `PostbackUrl` для определения страницы, которая должна запрашиваться с веб-сервера. Это свойство создает клиентский код JavaScript для запроса указанной страницы с помощью обработчика `onclick` клиентской стороны кнопки Submit:

```
<input type="submit" name="buttonSubmit" value="Submit"
  onclick="javascript:WebForm_DoPostBackWithOptions(
    new WebForm_PostBackOptions(&quot;buttonSubmit&quot;, &quot;&quot;, false,
      &quot;&quot;, &quot;ResultsPage.aspx&quot;, false, false))"
      id="buttonSubmit" />
```

Браузер отправляет все данные из формы в первой странице на новую страницу. Однако внутри только что запрошенной страницы необходимо извлечь данные из элементов управления, которые определены предыдущей страницей. Для доступа к элементам управления предыдущей страницы в классе `Page` определено свойство `PreviousPage`. Это свойство возвращает объект `Page`, в котором можно получать доступ к элементам управления этой страницы посредством метода `FindControl()`. Метод `FindControl()` определен так, чтобы возвращать объект `Control`, поэтому возвращаемое значение необходимо привести к типу искомого элемента управления:

```
DropDownList dropDownListEvents =
  (DropDownList)PreviousPage.FindControl("dropDownListEvents")
```

Фрагмент кода ***ResultsPage.aspx.cs***

Вместо использования метода `FindControl()` для доступа к значениям предыдущей страницы можно использовать строгую типизацию, что снижает вероятность ошибок во время разработки. Для обеспечения такой возможности в следующем практическом занятии мы определим нестандартную структуру, содержащую свойство из класса `default_aspx`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание строго типизированной предыдущей страницы

1. Добавьте в проект элемент нового класса `RegistrationInfo`, выбрав пункт меню Project⇨Add New Class (Проект⇨Добавить новый класс).
2. Реализуйте класс `RegistrationInfo`, как показано ниже:

```
public class RegistrationInfo
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Email { get; set; }
   public string SelectedEvent { get; set; }
}
```

*Фрагмент кода **RegistrationInfo.cs***

3. Добавьте общедоступное свойство `RegistrationInfo` в класс `Registration` в файле `Registration.aspx.cs`:

```
public RegistrationInfo RegistrationInfo
{
   get
   {
      return new RegistrationInfo
      {
         FirstName = textFirstName.Text,
         LastName = textLastName.Text,
         Email = textEmail.Text,
         SelectedEvent = dropDownListEvents.SelectedValue
      };
   }
}
```

*Фрагмент кода **Registration.aspx.cs***

4. Добавьте директиву `PreviousPageType` после директивы `Page` в файле `ResultPage.aspx`:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ResultsPage.aspx.cs"
    Inherits="EventRegistrationWeb.ResultsPage" %>
<%@ PreviousPageType VirtualPath="~/Registration.aspx" %>
```

*Фрагмент кода **ResultPage.aspx***

5. Теперь можно упростить код внутри метода `Page_Load()` класса `ResultsPage`:

```
protected void Page_Load(object sender, EventArgs e)
{
   try
   {
      RegistrationInfo ri = PreviousPage.RegistrationInfo;
      labelResult.Text = String.Format("{0} {1} selected the event {2}",
          ri.FirstName, ri.LastName, ri.SelectedEvent);
   }
   catch
   {
      labelResult.Text = "The originating page must contain " +
           "textFirstName, textLastName, textEmail controls";
   }
}
```

*Фрагмент кода **ResultPage.aspx.cs***

Описание работы

Директива `PreviousPageType` создает свойство типа `PreviousPage`, которое возвращает связанный с директивой тип. Внутри его реализации вызывается свойство `PreviousPage` базового класса, как показано в следующем фрагменте кода:

```
public new EventRegistrationWeb.Default PreviousPage {
   get {
      return ((EventRegistrationWeb.Default)(base.PreviousPage));
   }
}
```

Вместо использования атрибута `VirtualPath` директивы `PreviousPageType` для определения типа предыдущей страницы можно применять атрибут `TypeName`. Это удобно, если допускается существование нескольких предшествующих страниц. В этом случае необходимо определить базовый класс для всех предшествующих страниц и присвоить базовый класс атрибуту `TypeName`.

AJAX-обратная отправка ASP.NET

При обычной обратной отправке ASP.NET выполняется запрос всей страницы. При обратной отправке той же страницы, которую пользователь уже загрузил, обратная отправка снова возвращает всю страницу. Для уменьшения объема сетевого трафика можно осуществлять обратную отправку ASP.NET Ajax. При обратной отправке Ajax только часть страницы возвращается и обновляется с помощью JavaScript. Это легко реализовать с помощью `UpdatePanel`.

Для упрощения сравнения обратной отправки ASP.NET и обратной отправки Ajax в следующем практическом занятии мы запишем в метку текущее время при использовании и без использования элемента управления `UpdatePanel`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование элемента управления `UpdatePanel`

1. Откройте в Visual Studio ранее созданный проект `EventRegistrationWeb`.
2. В существующий веб-сайт добавьте новую веб-форму AJAX, назначив ей имя `UpdatePanelDemo.aspx`.
3. Добавьте к странице элемент `UpdatePanel`, находящийся в категории AJAX Extensions (Расширения Ajax) панели инструментов.
4. Добавьте элементы управления `Label` и `Button` внутрь `UpdatePanel` и еще по одному элементу управления `Label` и `Button` за пределами `UpdatePanel`. Установите свойство `Text` элемента управления `Button` внутри `UpdatePanel` в `AJAX Postback`, а свойство `Text` элемента управления `Button` вне `UpdatePanel` — в `ASP.NET Postback`.

```
<form id="form1" runat="server">
<div>
   <asp:ScriptManager ID="ScriptManager1" runat="server">
   </asp:ScriptManager>
</div>
<asp:UpdatePanel runat="server">
   <ContentTemplate>
      <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
      <asp:Button ID="Button1" runat="server" Text="AJAX Postback"
         OnClick="OnButtonClick" />
   </ContentTemplate>
</asp:UpdatePanel>
```

```
<asp:Label ID="Label2" runat="server" Text="Label"></asp:Label>
<asp:Button ID="Button2" runat="server" Text="ASP.NET Postback" />
</form>
```

Фрагмент кода ***UpdatePanelDemo.aspx***

5. Назначьте обработчик события `Click` по имени `OnButtonClick()` обеим кнопкам и реализуйте его, как показано ниже:

```
protected void OnButtonClick(object sender, EventArgs e)
{
  DateTime now = DateTime.Now;
  Label1.Text = now.ToLongTimeString();
  Label2.Text = now.ToLongTimeString();
}
```

Фрагмент кода ***UpdatePanelDemo.aspx.cs***

6. Запустите приложение и щелкните на обеих кнопках. Щелчок на кнопке AJAX Postback (Обратная отправка AJAX) обновляет только первую метку. В то же время щелчок на кнопке ASP.NET Postback (Обратная отправка ASP.NET) приводит к обновлению всей страницы (рис. 18.10).

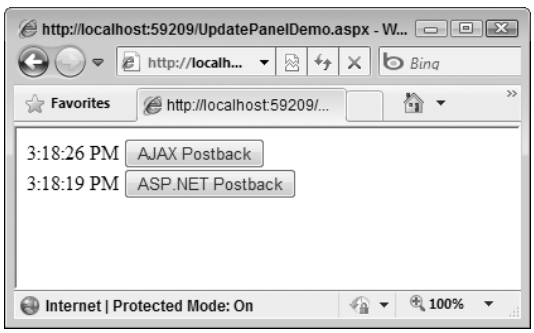


Рис. 18.10. *Элемент управления* **`UpdatePanel`** *в работе*

Описание работы

AJAX-страница ASP.NET требует наличия объекта `ScriptManager`. Его добавляют с использованием шаблона AJAX Web Form. Класс `ScriptManager` загружает функции JavaScript, реализующие несколько средств. Этот класс можно также применять для загрузки собственных пользовательских сценариев.

Свойства класса `ScriptManager` описаны в табл. 18.2.

Элементы управления `Button` в ASP.NET на странице ведут к созданию HTML-кнопок `Submit` (Отправить) на стороне клиента. `Button2` выполняет обычный HTTP-запрос `POST` к серверу. Поскольку кнопка `Button1` находится внутри `UpdatePanel`, для выполнения Ajax-запроса `POST` клиентский сценарий связывается с событием `Click` кнопки. Для отправки запроса серверу Ajax-запрос `POST` использует объект `XmlHttpRequest`. Сервер возвращает только те данные, которые требуются для обновления пользовательского интерфейса. Эти данные интерпретируются, и код JavaScript модифицирует элементы управления HTML внутри `UpdatePanel` в соответствии с новым пользовательским интерфейсом.

Страница может содержать несколько панелей `UpdatePanel`. Нужно просто добавить их на страницу, и каждая панель `UpdatePanel` будет обновляться при Ajax-запросе `POST`. Обновления могут управляться триггерами. Эта функциональность будет реализована в следующем практическом занятии.

Таблица 18.2. Свойства класса `ScriptManager`

Свойство	Описание
`EnablePageMethods`	Указывает, должны ли общедоступные статические методы, определенные в странице ASPX, быть доступными для вызова из клиентского сценария как методы веб-службы
`EnablePartialRendering`	Для обеспечения частичной визуализации с помощью `UpdatePanel` это свойство должно быть установлено в `true` (значение по умолчанию)
`LoadScriptsBeforeUI`	Определяет позицию вставки сценариев в возвращенной HTML-странице. Помещение сценариев внутрь элемента `<head>` ведет к их загрузке перед загрузкой интерфейса пользователя
`ScriptMode`	Указывает на необходимость использования отладочной или рабочей версии сценария
`ScriptPath`	Указывает корневой маршрут каталога размещения пользовательских сценариев
`Scripts`	Содержит коллекцию файлов пользовательских сценариев, которые должны визуализироваться на стороне клиента
`Services`	Содержит коллекцию ссылок веб-служб, которые могут вызываться из клиентского сценария

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Панель обновления с триггерами

1. Откройте в Visual Studio ранее созданный проект `EventRegistrationWeb`.
2. Добавьте к существующему веб-сайту новую веб-форму AJAX, назначив ей имя `UpdatePanelWithTrigger.aspx`.
3. Добавьте два элемента управления `UpdatePanel`.
4. В каждый элемент управления `UpdatePanel` добавьте элементы управления `Label` и `Button`.
5. Присвойте методу `OnButtonClick()` обработчик события `Click` обоих элементов управления `Button` и реализуйте метод, как показано ниже.

```
protected void OnButtonClick(object sender
{
  DateTime now = DateTime.Now;
  Label1.Text = now.ToLongTimeString();
  Label2.Text = now.ToLongTimeString();
}
```

Фрагмент кода ***UpdatePanelWithTrigger.aspx.cs***

6. Запустите приложение. Обе метки будут изменяться независимо от того, на каком элементе управления `Button` выполнен щелчок (рис. 18.11).
7. Измените свойство `UpdateMode` обоих элементов управления `UpdatePanel` с `Always` (всегда) на `Conditional` (условно).
8. Снова запустите приложение. Теперь изменяется только элемент управления `Label`, расположенный внутри элемента `UpdatePanel`, в котором был выполнен щелчок на элементе управления `Button`.

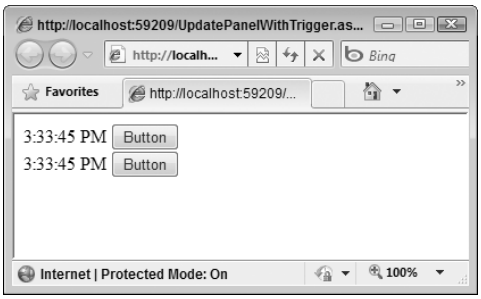


***Рис. 18.11**. Использование двух панелей* `UpdatePanel`

9. Выберите первый элемент `UpdatePanel` и щелкните на троеточии возле свойства `Triggers`. Откроется диалоговое окно UpdatePanelTrigger Collection Editor (Редактор коллекции `UpdatePanelTrigger`), показанное на рис. 18.12. Добавьте триггер `AsynchronousPostback`, установите свойство `ControlID` кнопки второго элемента `UpdatePanel` в `Button2`, а `EventName` — в `Click`.

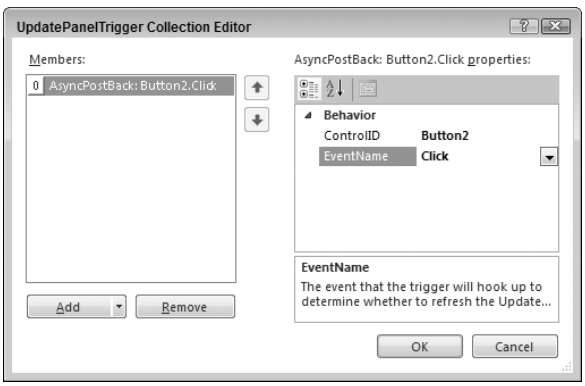


***Рис. 18.12**. Диалоговое окно UpdatePanelTrigger Collection Editor*

10. Запустите приложение. Щелчок на кнопке `Button2` приводит к обновлению содержимого обоих элементов управления `UpdatePanel`. Щелчок на кнопке `Button1` вызывает обновление содержимого только первого элемента `UpdatePanel`.

Описание работы

Поведение элемента управления `UpdatePanel` при обновлении можно изменять. По умолчанию он обновляется при каждой обратной отправке Ajax. Обновление можно изменить так, чтобы элементы управления обновлялись либо при обновлении из панели, либо при его запуске из элементов управления, расположенных вне панели.

ASPX-код определения триггера `AsyncPostbackTrigger` для `UpdatePanel1` приведен в следующем фрагменте.

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server" UpdateMode="Conditional">
    <ContentTemplate>
      <asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
      <asp:Button ID="Button1" runat="server" Text="Button"
        OnClick="OnButtonClick" />
    </ContentTemplate>
```

```
    <Triggers>
      <asp:AsyncPostBackTrigger ControlID="Button2" EventName="Click" />
    </Triggers>
</asp:UpdatePanel>
```

Фрагмент кода ***UpdatePanelWithTrigger.aspx***

Свойства элемента управления `UpdatePanel` описаны в табл. 18.3.

Таблица 18.3. Свойства класса `UpdatePanel`

Свойство	Описание
`ChildrenAsTriggers`	Когда это свойство установлено в значение `true`, содержимое элемента управления `UpdatePanel` обновляется при выполнении обратной отправки его дочерними элементами управления
`RenderMode`	Определяет режим визуализации панели. Возможные значения — `UpdatePanelRenderMode.Block` и `UpdatePanelRenderMode.Inline`. Значение перечисления `Block` указывает о необходимости визуализации дескриптора `<div>`, а `Inline` — о визуализации дескриптора `<span>`
`UpdateMode`	Выполняет установку в одно из значений перечисления `UpdatePanelUpdateMode`. Значение `Always` указывает на обновление панели при каждой обратной отправке Ajax, а `Conditional` — только в зависимости от триггеров
`Triggers`	Указывает коллекцию элементов `AsyncPostbackTrigger` и `PostbackTrigger` для определения того, когда должно выполняться обновление содержимого панели

Проверка достоверности ввода

Когда пользователи вводят данные, необходимо выполнять проверку для подтверждения их достоверности. Проверка может осуществляться на клиенте и на сервере. Проверка данных на клиенте может выполняться с помощью JavaScript. Однако если данные проверяются на клиенте с помощью JavaScript, они должны проверяться и на сервере, поскольку клиенту никогда нельзя доверять полностью. JavaScript в браузере можно отключить, и хакеры могут использовать другие функции JavaScript, которые принимают неверный ввод. Поэтому обязательно нужно проверять данные на сервере. Проверка данных на стороне клиента обеспечивает также более высокую производительность, поскольку не приходится дожидаться передачи данных по замкнутому контуру до тех пор, пока данные будут проверены.

ASP.NET исключает необходимость самостоятельного создания функций проверки. Существует множество элементов управления проверки данных, которые создают проверку на стороне клиента и сервера.

В следующем примере показан элемент управления проверкой `RequiredFieldValidator`, который связан с текстовым полем `textFirstname`. Все элементы управления проверкой достоверности имеют свойства `ErrorMessage` и `ControlToValidate`. Если введенные данные не верны, свойство `ErrorMessage` определяет отображаемое при этом сообщение. По умолчанию сообщение об ошибке отображается в позиции размещения элемента управления проверкой. Свойство `ControlToValidate` определяет элемент управления, в котором проверяются введенные данные.

```
<asp:TextBox ID="textFirstname" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
   ErrorMessage="Enter your first name" ControlToValidate="textFirstName">
</asp:RequiredFieldValidator>
```

Все элементы управления проверкой достоверности описаны в табл. 18.4.

Таблица 18.4. Элементы управления проверкой достоверности

Элемент управления	Описание
`RequiredFieldValidator`	Указывает, что вводимые данные являются обязательными для проверяемого элемента управления. Если проверяемый элемент управления имеет начальное установленное значение, которое пользователь должен изменять, это начальное значение может быть установлено в свойстве `InitialValue` элемента управления проверкой
`RangeValidator`	Определяет минимальное и максимальное значения, которые разрешено вводить пользователю. Особыми свойствами элемента управления являются `MinimumValue` и `MaximumValue`
`RegularExpressionValidator`	Свойство `ValidationExpression` позволяет устанавливать для проверки вводимых пользователем данных регулярное выражение, использующее синтаксис Perl 5
`CompareValidator`	Это свойство сравнивает несколько значений (таких как пароли). Это свойство проверки поддерживает не только сравнение двух значений на предмет их равенства, но и установку дополнительных опций с помощью свойства `Operator`. Свойство `Operator` типа `ValidationCompareOperator` определяет значения перечисления, такие как `Equal` (равно), `NotEqual` (не равно), `GreaterThan` (больше) и `DataTypeCheck` (проверка типа данных). `DataTypeCheck` позволяет проверить соответствие вводимых данных конкретному типу, например, для проверки достоверности ввода значения даты
`CustomValidator`	Если другие элементы управления проверкой не удовлетворяют требованиям проверки, можно использовать свойство `CustomValidator`. Оно позволяет определить функции проверки как на клиентской, так и на серверной стороне
`ValidationSummary`	Выводит сводку по странице вместо вывода сообщений об ошибках непосредственно в элементах управления вводом.

Созданный к этому моменту пример приложения позволяет пользователям вводить имя, фамилию и адрес электронной почты. В следующем практическом занятии мы расширим возможности приложения, применив в нем элементы управления проверки достоверности вводимых данных.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Проверка наличия обязательных данных и адреса электронной почты

1. С помощью Visual Studio откройте ранее созданный проект `EventRegistrationWeb`.
2. Откройте файл `Registration.aspx`.
3. Добавьте в таблицу новый столбец, выбрав правый столбец в представлении визуального конструктора и затем пункт меню Table⇨Insert⇨Column to the Right (Таблица⇨Вставить⇨Столбец справа).

4. Поля имени, фамилии и адреса электронной почты являются обязательными для ввода. В ходе проверки выполняется проверка соответствия адреса электронной почты принятому синтаксису. Добавьте три элемента управления `RequiredFieldValidator` и один элемент управления `RegularExpressionValidator`, как показано на рис. 18.13.

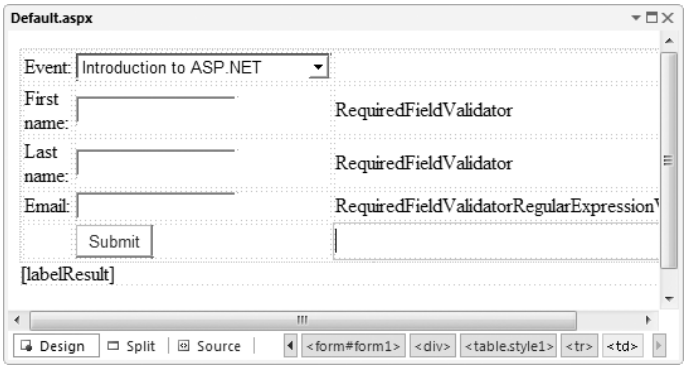


***Рис. 18.13**. Добавление элементов управления проверкой достоверности*

5. Сконфигурируйте элементы управления проверкой, как показано в следующей таблице.

Элемент управления	Свойство	Значение
`RequiredFieldValidator`	`ErrorMessage`	`First name is required.`
	`ControlToValidate`	`textFirstName`
`RequiredFieldValidator`	`ErrorMessage`	`Last name is required.`
	`ControlToValidate`	`textLastName`
`RequiredFieldValidator`	`ErrorMessage`	`Email is required.`
	`ControlToValidate`	`textEmail`
	`Display`	`Dynamic`
`RegularExpressionValidator1`	`ErrorMessage`	`Enter a valid email.`
	`ControlToValidate`	`textEmail`
	`ValidationExpression`	`\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*`
	`Display`	`Dynamic`

6. Регулярное выражение не обязательно вводить вручную. Вместо этого можно щелкнуть на кнопке с многоточием возле свойства `ValidationEpression` в окне Properties, чтобы запустить редактор Regular Expression Editor (Редактор регулярных выражений), показанный на рис. 18.14. Этот редактор предоставляет ряд предварительно определенных регулярных выражений, в том числе для проверки адресов электронной почты.

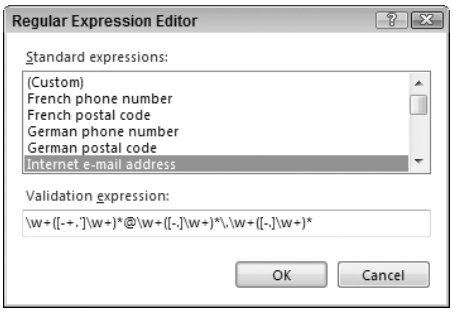


***Рис. 18.14**. Редактор регулярных выражений*

7. Если обратная отправка выполняется на страницу, которая отличается от содержащей элементы управления проверкой достоверности (с помощью установленного ранее свойства `PostBackUrl`), на новой странице с помощью свойства `IsValid` нужно проверить достоверность результата предыдущей страницы. Добавьте следующий код в метод `Page_Load()` класса `ResultsPage`:

```
protected void Page_Load(object sender, EventArgs e)
{
  try
  {
    if (!PreviousPage.IsValid)
    {
      labelResult.Text = "Error in previous page";
      return;
    }
    //...
```

*Фрагмент кода **ResultsPage.aspx.cs***

8. Теперь приложение можно запустить. Если данные не введены или введены неправильно, элементы управления проверкой отображают сообщения об ошибках, как показано на рис. 18.15.

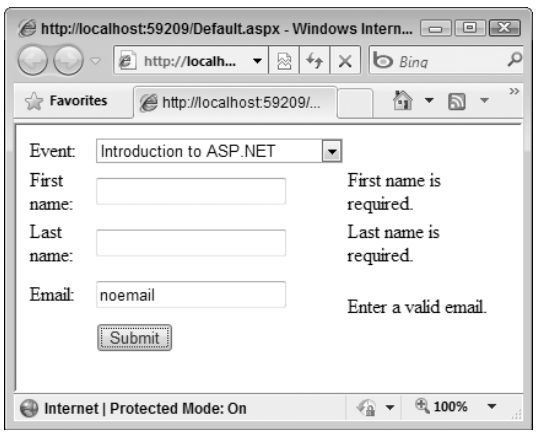


***Рис. 18.15**. Работа элементов управления проверкой достоверности*

Описание работы

Элементы управления проверкой достоверности создают код JavaScript клиентской стороны для проверки ввода на клиенте, а также код серверной стороны для выполнения проверки на сервере. Устанавливая свойство `EnableClientScript` элемента управления проверкой в `false`, JavaScript можно отключить. Вместо того чтобы изменять свойство для каждого элемента управления проверкой, JavaScript можно отключить, установив свойство `ClientTarget` класса `Page`.

В зависимости от типа клиента элементы управления ASP.NET могут возвращать код JavaScript клиенту. Это поведение определяется свойством `ClientTarget`. По умолчанию для этого свойства установлена строка `"automatic"`. В этом случае, в зависимости от функциональных возможностей веб-браузера, код сценария либо возвращается, либо нет. Если `ClientTarget` установлено в значение `"downlevel"`, код сценария не возвращается ни для каких клиентов, в то время как установка свойства в `"uplevel"` обеспечивает возврат кода сценария во всех случаях.

Установка свойства `ClientTarget` может быть выполнена внутри метода `Page_Load()` класса `Page`:

```
protected void Page_Load(object sender, EventArgs e)
{
   ClientTarget = "downlevel";
}
```

Управление состоянием

Протокол HTTP не поддерживает состояние. Подключение, инициированное от клиента к серверу, может быть закрыто после выполнения любого запроса. Однако обычно при переходе от одной страницы к другой требуется запоминание определенной информации о клиенте. Существует несколько способов решения этой задачи.

Основное различие между способами сохранения состояния связано с тем, где сохраняется состояние — на клиенте или на сервере. Краткие описания технологий управления состоянием и сроки действия состояния приведены в табл. 18.5.

Таблица 18.5. Технологии управления состоянием

Тип состояния	Принадлежность ресурса	Срок действия
Состояние представления	Клиент	В пределах только одной страницы
Cookie-наборы	Клиент	Временные cookie-наборы удаляются при закрытии браузера. Постоянные cookie-наборы сохраняются на диске клиентской системы
Сеанс	Сервер	Состояние сеанса связано с состоянием браузера. Состояние сеанса становится недействительным по истечении интервала ожидания (по умолчанию — 20 минут)
Приложение	Сервер	Состояние приложения является общим для всех клиентов. Оно остается действующим до перезапуска сервера
Кэш	Сервер	Аналогично состоянию приложения, кэш является ресурсом совместного использования. Однако в тех случаях, когда кэш должен быть объявлен недействительным, это средство управления оказывается значительно более эффективным

Эти технологии более подробно рассматриваются в последующих разделах.

Управление состоянием на стороне клиента

В этом разделе мы приступаем к ознакомлению с особенностями управления состоянием на стороне клиента и рассмотрим две технологии: управление состоянием представления и cookie-наборы.

Состояние представления

Мы уже обсуждали одну технологию сохранения состояния на стороне клиента — состояние представления. Состояние представления автоматически используется серверными веб-элементами управления для выполнения действий, обусловленных событиями. Состояние представления содержит ту же информацию о состоянии, которая содержалась в элементе управления во время отправки данных клиенту. Когда браузер возвращает форму серверу, состояние представления содержит исходные значения, но отправляемые элементы управления содержат новые значения. В случае если есть различия, вызываются соответствующие обработчики событий.

Недостаток использования состояния представления состоит в том, что данные всегда передаются от сервера клиенту и обратно, что ведет к увеличению объема сетевого трафика. Для уменьшения объема сетевого трафика состояние представления можно отключить. Чтобы выполнить это для всех элементов управления внутри страницы, с помощью директивы `Page` установите свойство `EnableViewState` в `false`:

```
<%@ Page Language="C#" AutoEventWireUp="true" CodeFile="Default.aspx.cs"
    Inherits="Default" EnableViewState="false" %>
```

Состояние представления можно конфигурировать также в элементе управления, устанавливая его свойство `EnableViewState`. Когда свойство `EnableViewState` определено в элементе управления, его значение используется независимо от значения, установленного в конфигурации страницы. Значение конфигурации страницы применяется только для соответствующих элементов управления, когда состояние представления не сконфигурировано.

Внутри состояния представления можно хранить также индивидуальные данные. Для этого необходимо использовать индексатор применительно к свойству `ViewState` класса `Page`. Имя, применяемое для доступа к значению состояния представления, можно определить с помощью аргумента `index`:

```
ViewState["mydata"] = "my data";
```

Ранее сохраненное состояние представления можно прочитать следующим образом:

```
string mydata = (string)ViewState["mydata"];
```

В HTML-разметке, отправляемой клиенту, состояние представления всей страницы можно увидеть в скрытом поле:

```
<input type="hidden" name="__VIEWSTATE"
       value="/wEPDwUKLTU4NzY5NTcwNw8WAh4HbXlzdGF0ZQUFbXl2YWwWAgIDD2QWAg
           IFDw8WAh4EVGV4dAUFbXl2YWxkZGTCdCywUOcAW97aKpcjt1tzJ7ByUA==" />
```

Использование скрытых полей обеспечивает преимущество в том, что каждый браузер может использовать данную функциональную возможность, и пользователь не может ее отключить.

Состояние представления запоминается только внутри данной страницы. Если требуется, чтобы состояние распространялось на различные страницы, для сохранения информации о состоянии на клиентском компьютере нужно применять cookie-наборы.

Cookie-наборы

Cookie-набор определяется в HTTP-заголовке. Для отправки cookie-набора клиенту служит класс `HttpResponse`. В классе `Page` имеется свойство `Response`, которое возвращает объект типа `HttpResponse`. В классе `HttpResponse` определено свойство `Cookies`, которое возвращает объект `HttpCookieCollection`. Посредством `HttpCookieCollection` клиенту можно вернуть несколько cookie-наборов.

В приведенном ниже примере кода показано, как cookie-набор может быть отправлен клиенту. Прежде всего, создается экземпляр объекта `HttpCookie`. В конструкторе этого класса устанавливается имя cookie-набора — в данном случае `mycookie`. Класс `HttpCookie` имеет свойство `Values`, предназначенное для добавления нескольких значений cookie-набора. Если нужно возвращать только одно значение cookie-набора, вместо `Values` можно использовать свойство `Value`. Однако если предполагается отправка нескольких значений cookie-наборов, лучше добавить значения в один cookie-набор, чем применять несколько cookie-наборов.

```
string myval = "myval";
var cookie = new HttpCookie("mycookie");
cookie.Values.Add("mystate", myval);
Response.Cookies.Add(cookie);
```

Cookie-наборы могут быть временными и действовать в течение сеанса браузера, или же они могут храниться на диске клиента. Чтобы сделать cookie-набор постоянным, нужно установить свойство `Expires` объекта `HttpCookie`. Дата, определенная в свойстве `Expires`, определяет конечный срок, по истечении которого cookie-набор становится недействительным. В следующем примере этот срок установлен равным трем месяцам, начиная от текущей даты.

```
var cookie = new HttpCookie("mycookie");
cookie.Values.Add("mystate", "myval");
cookie.Expires = DateTime.Now.AddMonths(3);
Response.Cookies.Add(cookie);
```

Хотя данное свойство позволяет устанавливать конкретную дату, нет никакой гарантии, что cookie-набор будет храниться вплоть до ее достижения. Пользователь может удалить cookie-набор, а приложение браузера удаляет cookie-наборы, если количество этих локально хранящихся данных слишком велико. Браузер ограничивает число cookie-наборов для одного сервера двадцатью, а для всех серверов — тремястами. По достижении предельного значения cookie-наборы, которые не использовались в течение некоторого времени, удаляются.

Когда клиент запрашивает страницу с сервера, а cookie-набор для этого сервера доступен на компьютере клиента, cookie-набор отправляется серверу в качестве составной части HTTP-запроса. Считывание cookie-набора в странице ASP.NET может быть выполнено обращением к коллекции `cookies` в объекте `HttpRequest`.

Подобно HTTP-ответу, класс `Page` имеет свойство `Request`, которое возвращает объект типа `HttpRequest`. Свойство `Cookies` возвращает объект `HttpCookieCollection`, который можно применять для чтения cookie-наборов, отправляемых клиентом. Обращение к cookie-набору может выполняться по его имени с индексатором, после чего свойство `Values` объекта `HttpCookie` используется для извлечения значения из cookie-набора:

```
HttpCookie cookie = Request.Cookies["mycookie"];
string myval = cookie.Values["mystate"];
```

Среда ASP.NET упрощает работу с cookie-наборами, но следует помнить о связанных с ними ограничениях. Помните, что браузер принимает только 20 cookie-наборов для одного сервера и 300 cookie-наборов для всех серверов. Кроме того, cookie-набор не может содержать более 4 Кбайт данных. Эти ограничения предотвращают переполнение диска клиента cookie-наборами.

Управление состоянием на стороне сервера

Вместо того чтобы запоминать состояние на системе клиента, его можно запомнить на сервере. Вспомните, что сохранение состояния на клиентской стороне сопряжено с тем недостатком, что это ведет к увеличению объема данных, пересылаемых по сети. Запоминание состояния на стороне сервера имеет тот недостаток, что сервер должен выделять ресурсы для своих клиентов. Технологии управления состоянием на серверной стороне рассматриваются в последующих разделах.

Состояние сеанса

Состояние сеанса связано с состоянием браузера. Сеанс запускается, когда клиент впервые открывает страницу ASP.NET на сервере, и завершается, если клиент не обращается к серверу в течение 20 минут.

Внутри глобального класса приложения можно определить собственный код, который должен выполняться при запуске или завершении сеанса. Для создания такого класса выберите пункт меню Project⇨Add New Item⇨Global Application Class (Проект⇨Добавить новый элемент⇨Глобальный класс приложения). При создании этого класса создается файл `global.asax`. Внутри этого файла в классе `Global`, производном от базового класса `HttpApplication`, определяется набор подпрограмм-обработчиков.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.SessionState;

namespace EventRegistrationWeb
{
  public class Global : System.Web.HttpApplication
  {
    protected void Application_Start(object sender, EventArgs e)
    {
      // Код, который выполняется при запуске приложения.
    }
    protected void Session_Start(object sender, EventArgs e)
    {
      // Код, который выполняется при запуске нового сеанса.
    }
    protected void Application_BeginRequest(object sender, EventArgs e)
    {
    }
    protected void Application_AuthenticateRequest(object sender, EventArgs e)
    {
    }
    protected void Application_Error(object sender, EventArgs e)
    {
      // Код, который выполняется при возникновении необработанной ошибки.
    }
    protected void Session_End(object sender, EventArgs e)
    {
      // Код, который выполняется при завершении сеанса.
      // Примечание. Событие Session_End генерируется, только если режим
      // состояния сеанса установлен в InProc в файле Web.config. Если режим
      // сеанса установлен в StateServer или SQLServer, событие не генерируется.
    }
```

```
        protected void Application_End(object sender, EventArgs e)
        {
           // Код, который выполняется при останове приложения.
        }
     }
}
```

Фрагмент кода ***Global.asax.cs***

Состояние сеанса может храниться внутри объекта `HttpSessionState`. Доступ к объекту состояния сеанса, связанному с текущим HTTP-контекстом, может осуществляться с помощью свойства `Session` класса `Page`. Переменные сеанса могут инициализироваться в обработчике события `Session_Start()`. В следующем примере состояние сеанса, названное `mydata`, инициализируется в значение `0`:

```
void Session_Start(Object sender, EventArgs e) {
   // Код, выполняемый при запуске приложения
   Session["mydata"] = 0;
}
```

В следующем примере показано, как прочитать состояние сеанса с использованием свойства `Session` и имени состояния сеанса:

```
void Button1_Click(object sender, EventArgs e)
{
  int val = (int)Session["mydata"];
  Label1.Text = val.ToString();
  val += 4;
  Session["mydata"] = val;
}
```

Чтобы связать клиента с его переменными сеанса, по умолчанию среда ASP.NET использует временный cookie-набор с соответствующим идентификатором сеанса. В ASP.NET поддерживаются также сеансы без cookie-наборов, в которых идентификаторы URL-адресов служат для отображения HTTP-запросов к одному сеансу.

Состояние приложения

Если данные должны совместно использоваться различными клиентами, нужно использовать состояние приложения. Работа с состоянием приложения аналогична работе с состоянием сеанса. При этом применяется класс `HttpApplicationState`, доступ к которому можно получить через свойство `Application` класса `Page`.

В следующем примере переменная приложения по имени `userCount` инициализируется при запуске веб-приложения. `Application_Start()` — это метод обработчика события в файле `global.asax`, который вызывается при запуске первой страницы ASP.NET веб-сайта. Эта переменная используется для учета каждого пользователя, обращающегося к веб-сайту:

```
void Application_Start(Object sender, EventArgs e) {
   // Код, который выполняется при запуске приложения.
   Application["userCount"] = 0;
}
```

В обработчике события `Session_Start()` значение переменной приложения `userCount` увеличивается на единицу. Перед изменением переменной приложения объект приложения должен быть заблокирован методом `Lock()`. В противном случае возможно возникновение проблем многопоточности, поскольку несколько клиентов могут обращаться к переменной приложения одновременно. После изменения значения переменной приложения необходимо вызвать метод `Unlock()`. Имейте в виду, что временной интервал между блокированием и разблокированием очень краток — за это время нельзя выполнить считывание файлов или информации из базы данных. В противном случае другим клиентам придется дожидаться окончания обращения к данным.

```
void Session_Start(Object sender, EventArgs e) {
   // Код, который выполняется при запуске нового сеанса.
   Application.Lock();
   Application["userCount"] = (int)Application["userCount"] + 1;
   Application.UnLock();
}
```

Чтение данных из состояния приложения осуществляет столь же просто, как их считывание из состояния сеанса:

```
Label1.Text = this.Application["userCount"].ToString();
```

Не храните в состоянии приложения слишком большой объем данных, поскольку состояние приложения занимает ресурсы сервера до тех пор, пока он не будет остановлен или перезапущен.

Кэш

Кэш — это состояние серверной стороны, которое подобно состоянию приложения в том смысле, что оно совместно используется всеми клиентами. Кэш отличается от состояния приложения большей гибкостью: существует множество возможностей определения того, когда состояние должно становиться недействительным. Вместо считывания файла при каждом запросе или чтения базы данных данные могут сохраняться внутри кэша.

Для работы с кэшем необходимо пространство имен `System.Web.Caching` и класс `Cache`. Добавление объекта в кэш показано в следующем примере:

```
Cache.Add("mycache", myobj, null, DateTime.MaxValue,
   TimeSpan.FromMinutes(10), CacheItemPriority.Normal, null);
```

Класс `Page` имеет свойство `Cache`, которое возвращает объект `Cache`. С помощью метода `Add()` класса `Cache` в кэш можно добавить любой объект. Первый параметр метода `Add()` определяет имя элемента кэша. Второй параметр — это объект, который должен быть кэширован. С помощью третьего параметра можно определять зависимости, например, элемент кэша может зависеть от какого-то файла. При изменении файла объект кэша становится недействительным. В приведенном примере какие-либо зависимости отсутствуют, поскольку в третьем параметре передается `null`.

Четвертый и пятый параметры позволяют указывать продолжительность времени, в течение которого элемент кэша остается действительным. Четвертый параметр определяет абсолютное значение времени, когда элемент кэша будет объявлен недействительным, а пятый параметр задает скользящий временной интервал, по истечении которого элемент кэша становится недействительным. В приведенном примере используется скользящий временной интервал, в соответствии с которым элемент кэша признается недействительным по истечении 10 минут.

Шестой параметр определяет приоритет кэша, указываемый с помощью перечисления `CacheItemPriority`. Если рабочий процесс ASP.NET интенсивно использует память, исполняющая среда ASP.NET удаляет элементы кэша в соответствии с их приоритетами. Элементы с более низким приоритетом удаляются первыми. Последний параметр позволяет определить метод, который должен вызываться при удалении элемента кэша. Пример использования этого параметра — ситуация, когда кэш зависит от файла. При изменении файла элемент кэша удаляется и вызывается обработчик события. С помощью обработчика события кэш может быть перезагружен за счет повторного чтения файла.

Элементы кэша могут считываться с помощью индексатора, как это имело место с состояниями сеанса и приложения. Прежде чем использовать объект, возвращенный свойством `Cache`, всегда следует проверять, не равен ли результат `null`, что происходит, когда кэша становится недействительным. Когда значение, возвращенное из индексатора `Cache`, отлично от `null`, возвращенный объект может быть приведен к типу, который применялся для сохранения элемента в кэше:

```
object o = Cache["mycache"];
if (o == null)
{
   // Перезагрузка кэша.
}
else
{
    // Использование кэша.
   MyClass myObj = (MyClass)o;
   //...
}
```

Стили

В среде Visual Studio поддерживается стилизация веб-страниц с помощью каскадных таблиц стилей (Cascading Style Sheets – CSS). Таблицы CSS позволяют определять внешний вид и форматирование HTML-страниц. Вместо того чтобы индивидуально настраивать каждый HTML-элемент, с помощью CSS можно создать стили для конкретных элементов (что будет выполнено в следующем практическом занятии), а затем просто ссылаться на эти стили по имени.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Определение стилей элементов

1. Откройте ранее созданный проект веб-приложения `EventRegistrationWeb`.
2. Выбрав пункт меню Project⇨New Folder (Проект⇨Новая папка), создайте новую папку `Styles`.
3. Выделите эту папку в окне Solution Explorer и создайте новую таблицу стилей, выбрав пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент), а затем Style Sheet (Таблица стилей). Назначьте новой таблице стилей имя `Site.css`.
4. По умолчанию эта таблица стилей содержит пустой элемент `body`.
5. Щелкните внутри фигурных скобок элемента `body`, откройте контекстное меню и выберите в нем пункт Build Style (Построить стиль). Откроется диалоговое окно Modify Style (Изменение стиля), показанное на рис. 18.16.
6. Выберите категорию Font (Шрифт) и измените параметр `font-family` (семейство шрифтов) на `Arial, Helvetica, sans-serif`. Измените параметр `font-size` (размер шрифта) на `.80 em`, а параметр `color` (цвет) на `#FFFF00`.
7. В этом же диалоговом окне выберите категорию Background (Фон) и измените цвет фона на `#008080`.
8. Выберите категорию Box (Рамка) и измените параметры заполнения (`padding`) и поля (`margin`) на `0`.
9. Теперь таблица стилей должна выглядеть, как показано ниже:

```
body
{
   font-family: Arial, Helvetica, sans-serif;
   font-size: .80em;
   background-color: #008080;
   color: #FFFF00;
   padding: 0px;
   margin: 0px;
}
```

Фрагмент кода ***Site.css***

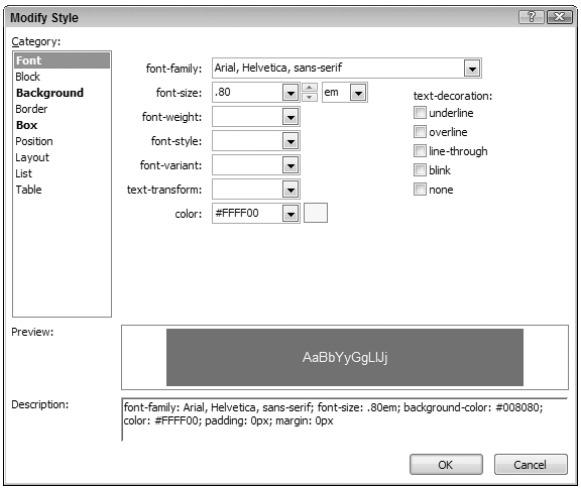


***Рис. 18.16**. Диалоговое окно Modify Style*

10. В представлении исходного кода редактора CSS выберите в контекстном меню пункт Add Style Rule (Добавить правило стиля) и в открывшемся диалоговом окне выберите элемент `a:hover`, как показано на рис. 18.17. Щелкните на кнопке OK.

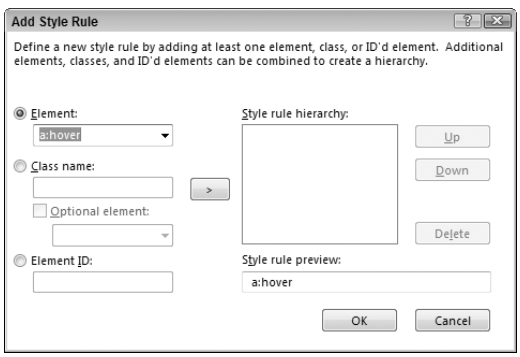


***Рис. 18.17**. Выбор элемента `a:hover`*

11. Выберите в контекстном меню пункт Build Style, чтобы снова отрыть диалоговое окно Modify Style. Выберите категорию Font, измените цвет на `#FF0000` и установите флажки `underline` (подчеркивание) и `overline` (надчеркивание) в группе `text-decoration` (декорирование текста). Результирующий CSS-код должен выглядеть подобно показанному ниже:

```
a:hover
{
   color: #FF0000;
   text-decoration: underline overline;
}
```

*Фрагмент кода **Site.css***

12. Добавьте правила стиля для элементов `a:active`, `a:link`, `a:visited` и `h1` в соответствии со следующим фрагментом кода:

```
a:link, a:visited
{
  color: #00FFFF;
}

a:active
{
  color: #00FFFF;
}

a:hover
{
  color: #FF0000;
  text-decoration: underline overline;
}
h1
{
  text-align: center;
}
```

Фрагмент кода ***Site.css***

13. Создайте новую веб-страницу `StylesDemo.aspx`. Перетащите файл `Site.css` из Solution Explorer в представление конструктора редактора. Цвет фона страницы немедленно изменится.

14. Перейдите к представлению исходного кода и удостоверьтесь, что новая запись `link` ссылается на созданную таблицу стилей:

```
<head runat="server">
  <title></title>
  <link href="Styles/Site.css" rel="stylesheet" type="text/css" />
</head>
```

Фрагмент кода ***StylesDemo.aspx***

15. Внутрь элемента `body` добавьте дескриптор `h1` и привяжите его к странице, как показано в следующем фрагменте:

```
<body>
  <form id="form1" runat="server">
  <h1>
    Styles Demo</h1>
  <div>
    <a href="http://www.wrox.com">Wrox Press</a>
  </div>
  </form>
</body>
```

Фрагмент кода ***StylesDemo.aspx***

16. Выбрав пункт меню Debug⇨Start without Debugging (Отладка⇨Запустить без отладки), запустите браузер, чтобы просмотреть страницу. Чтобы увидеть изменения цвета и оформления текста, проверьте применение стилей страницы к заголовку, ссылке и помещению курсора над ссылкой. На рис. 18.18 показаны настройки для категории Box диалогового окна Modify Style.

Описание работы

Поскольку ссылка на CSS-файл осуществляется посредством элемента ссылки, браузер запрашивает эту страницу вместе с HTML-разметкой. Затем браузер использует стилизованные элементы из CSS-файла для изменения внешнего вида HTML-элементов.

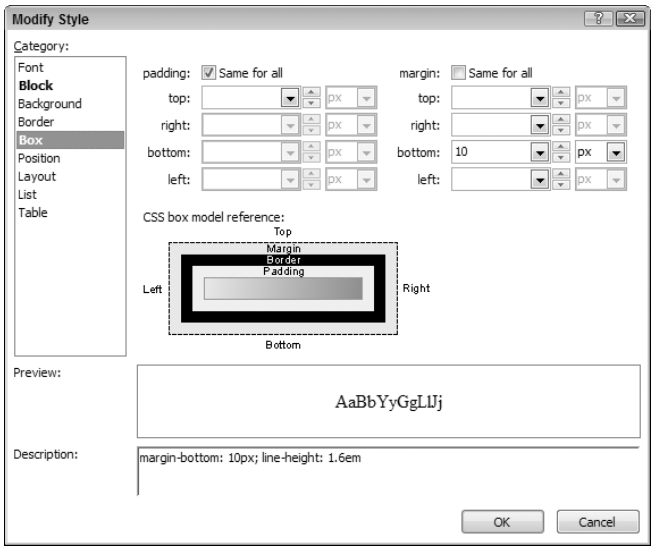


***Рис. 18.18**. Категория* Box *диалогового окна* Modify Style

С помощью CSS можно не только изменять стиль конкретных HTML-дескрипторов, но и определять классы, на которые производится ссылка из HTML-дескрипторов. Это будет выполнено в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Определение классов стилей

1. Откройте таблицу стилей `Site.css`.
2. Откройте диалоговое окно Add Style Rule и добавьте новый класс `bottom`. В диалоговом окне в области предварительного просмотра правила стиля имя класса будет содержать префикс в виде точки (`.`). Щелкните на кнопке OK.
3. Откройте диалоговое окно Modify Style.
4. Выберите категорию Font и укажите в качестве размера шрифта значение `x-small`.
5. Выберите категорию Block (Блок) и укажите для выравнивания по вертикали (`vertical-align`) значение `text-bottom` (по нижней границе текста), а для выравнивания текста (`text-align`) — `center` (по центру).
6. Выберите категорию Box и укажите для размеров всех полей `top` (верхнее), `right` (правое), `bottom` (нижнее) и `left` (левое) значение `5`.
7. Выберите категорию Position (Позиция) и определите высоту равной `40 px`.
8. Проверьте результат в файле `Site.css`.

```
.bottom
{
  margin: 5px;
  height: 40px;
  text-align: center;
  vertical-align: text-bottom;
  font-size: x-small;
}
```

Фрагмент кода ***Site.css***

9. Откройте файл StylesDemo.aspx и добавьте в него элемент div, содержащий следующий текст:

```
<div class="bottom">
   Copyright (c) 2010 Wrox Press
</div>
```

Фрагмент кода ***StylesDemo.aspx***

10. Запустите браузер, чтобы увидеть файл StylesDemo.aspx. Удостоверьтесь, что стиль для элемента div применен.

Описание работы

Вместо того чтобы определять стили каждого элемента в каждой странице веб-сайта, их можно определить в общем месте. Диалоговое окно Modify Style позволяет получить представление обо всех элементах, которые могут быть изменены. При открытии страницы браузер применяет стили и организует элементы соответствующим образом.

Некоторые стили по-разному применяются в различных браузерах. Обязательно проверьте внешний вид своей страницы во всех браузерах, которые должны поддерживаться.

Используя стили, можно не только применять размеры шрифтов и цвета, но и определять макет веб-страницы. Макет можно строить не только посредством стилей, но и с помощью HTML-таблиц. Но CSS не только обеспечивает большую гибкость в определении макета, чем HTML-таблицы, но и предоставляет ряд преимуществ в плане удобства доступа благодаря отделению содержимого от визуальной информации. Именно поэтому в последнее время предпочтение обычно отдают веб-дизайну без применения таблиц.

Поскольку настоящая книга посвящена программированию на C#, а не дизайну HTML-страниц, она содержит лишь краткие общие сведения о CSS. Для более подробного ознакомления с CSS обращайтесь к книге *HTML, XHTML и CSS. Библия пользователя* (издательство "Диалектика", 2010 г.).

Мастер-страницы

Большинство веб-сайтов повторно используют часть своего содержимого на каждой странице: очень часто такие элементы, как логотипы компаний и меню, доступны на каждой странице. Нет никакой нужды повторять общие элементы интерфейса пользователя на каждой странице. Вместо этого общие элементы могут быть помещены на *мастер-страницу* (master page). Мастер-страницы выглядят подобно обычным страницам ASP.NET, но определяют заполнители, которые замещаются *страницами содержимого*.

Мастер-страница имеет расширение файла .master и в первой строке файла содержит директиву Master, как показано в следующем примере:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="MasterPage.master.cs"
   Inherits="MasterPage" %>
```

Только мастер-страницы веб-сайта используют HTML-элементы <html>, <head>, <body> и <form>. Сами веб-страницы определяют только содержимое, которое вставляется внутрь заполнителей. Собственное содержимое веб-страницы могут помещать внутрь элемента управления ContentPlaceHolder. Мастер-страница может определять используемое по умолчанию содержимое для элемента ContentPlaceHolder на тот случай, если веб-страница его не определяет:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
    <asp:contentplaceholder id="head" runat="server">
    </asp:contentplaceholder>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:contentplaceholder id="ContentPlaceHolder1" runat="server">
      </asp:contentplaceholder>
    </div>
  </form>
</body>
</html>
```

Чтобы использовать мастер-страницу, в директиве `Page` необходимо применить атрибут `MasterPageFile`. Для замещения содержимого мастер-страницы применяется элемент управления `Content`. Этот элемент управления связывает элемент управления `ContentPlaceHolder` с элементом `ContentPlaceHolderID`.

```
<%Page Language="C#" MasterPageFile="~/MasterPage.master"
  AutoEventWireUp="true" CodeFile="Default.aspx.cs" Inherits="default"
  Title="Untitled Page" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head"
  Runat="Server"></asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="Server"></asp:Content>
```

Вместо того чтобы определять мастер-страницу с помощью директивы `Page`, с помощью элемента `<pages>` в файле `web.config` можно назначить всем веб-страницам стандартную мастер-страницу:

```
<configuration>
  <system.web>
    <pages masterPageFile="~/MasterPage.master">
      <!--...-->
    </pages>
  </system.web>
</configuration>
```

Если файл мастер-страницы сконфигурирован в `web.config`, страницы ASP.NET должны содержать в своем файле конфигурацию элемента `Content`, как показано в предшествующем примере. В противном случае атрибут `masterPageFile` использоваться не будет. При наличии и атрибута `MasterPageFile` директивы `Page`, и записи в файле `web.config` предпочтение отдается директиве `Page`. Это позволяет определять (в `web.config`) стандартный файл мастер-страницы, но замещать его для определенных веб-страниц.

Мастер-страницу можно также изменять программно. Это позволяет применять различные мастер-страницы для разных устройств или типов браузеров. Последнее место, где можно изменять мастер-страницу — метод обработчика `Page_PreInit`. В следующем примере кода свойство `MasterPageFile` класса `Page` устанавливается в `IE.master`, если вместе со своим именем браузер отправляет строку `MSIE` (что присуще Microsoft Internet Explorer), или в `Default.master` для всех других браузеров.

```
public partial class ChangeMaster: System.Web.UI.Page
{
  void Page_Load(object sender, EventArgs e)
  {
  }
```

```
    void Page_PreInit(object sender, EventArgs e)
    {
      if (Request.UserAgent.Contains("MSIE"))
      {
        this.MasterPageFile = "~/IE.master";
      }
      else
      {
        this.MasterPageFile = "~/Default.master";
      }
    }
}
```

В следующем практическом занятии будет создана собственная мастер-страница, которая имеет заголовок и тело, а основная часть мастер-страницы должна замещаться индивидуальными страницами.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание мастер-страницы

1. Откройте проект веб-приложения `EventRegistrationWeb`.
2. Добавьте новый элемент Master Page (Мастер-страница) и назовите его `Events.master`.
3. Перейдите в режим визуального конструктора и примените таблицу стилей `Site.css`. Перетащите файл `Site.css` из Solution Explorer в окно редактора.
4. Измените идентификатор (`ID`) второго элемента `ContentPlaceHolder` на `ContentPlaceHolderMain` и назначьте содержимое класса CSS элементу `div`, как показано ниже:

```
<div class="content">
  <asp:ContentPlaceHolder ID="ContentPlaceHolderMain" runat="server">
  </asp:ContentPlaceHolder>
</div>
```

*Фрагмент кода **Events.Master***

5. Добавьте следующие элементы `div` и `h1` перед ранее измененным элементом `div`:

```
<div class="header">
  <h1>
    Event Registration
  </h1>
</div>
<div class="navigation">
  Menu will go here
</div>
```

*Фрагмент кода **Events.Master***

6. Добавьте следующий элемент `div` после элемента `div`, содержащего `ContentPlaceHolder`:

```
<div class="bottom">
  Copyright (c) 2010 Wrox Press
</div>
```

*Фрагмент кода **Events.Master***

7. Полная страница должна выглядеть, как показано в следующем фрагменте:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Events.master.cs"
           Inherits="EventRegistrationWeb.Events" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
   <asp:ContentPlaceHolder ID="head" runat="server">
   </asp:ContentPlaceHolder>
   <link href="Styles/Site.css" rel="stylesheet" type="text/css" />
</head>

<body>
   <form id="form1" runat="server">
   <div class="header">
      <h1>
        Event Registration
      </h1>
   </div>
   <div class="navigation">
      Menu will go here
   </div>
   <div class="content">
      <asp:ContentPlaceHolder ID="ContentPlaceHolderMain" runat="server">
      </asp:ContentPlaceHolder>
   </div>
   <div class="bottom">
      Copyright (c) 2010 Wrox Press
   </div>
   </form>
</body>
</html>
```

Фрагмент кода ***Events.Master***

Описание работы

Как было отмечено ранее, мастер-страница содержит HTML-разметку, включая дескрипторы `<form>`, содержащие заполнители, в которых содержимое будет замещено страницами, использующими данную мастер-страницу. HTML-разметка совместно со связанной с ней таблицей стилей CSS определяет макет страницы. Только заполнители `ContentPlaceHolder` замещаются страницами содержимого. При необходимости замещения различных частей страницы можно определять несколько заполнителей содержимого.

После создания мастер-страницу можно использовать из веб-страницы, как показано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование мастер-страницы

1. Добавьте в веб-приложение новый элемент типа Web Form using Master Page (Веб-форма, использующая мастер-страницу) и назовите его `Default.aspx`.
2. Откроется диалоговое окно Select a Master Page (Выберите мастер-страницу), показанное на рис. 18.19. Выберите мастер-страницу `Events.master`. Щелкните на кнопке OK.
3. Представление исходного кода файла `Default.aspx` отображает только два элемента управления `Content`, расположенные после директивы `Page`, которая ссылается на элементы управления `ContentPlaceHolder` из мастер-страницы. Измените значения свойств `ID` элементов управления `Content` на `ContentHead` и `ContentMain`.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
         AutoEventWireup="true" CodeBehind="Default.aspx.cs"
         Inherits="EventRegistrationWeb.Default" %>
<asp:Content ID="ContentHead" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content ID="ContentMain" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
</asp:Content>
```

Фрагмент кода **Default.aspx**

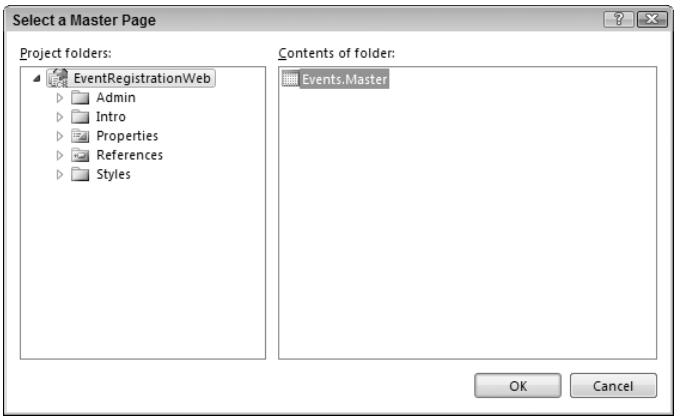


Рис. 18.19. *Диалоговое окно* Select a Master Page

4. В среде Visual Studio перейдите к представлению конструктора. Это представление отображает содержимое мастер-страницы, которое нельзя изменять из страницы, в том числе заголовок и сведения об авторских правах. Введите какой-то текст в элементе управления `Content` и выровняйте его по центру.
5. Перейдите к представлению исходного кода, которое отобразит код, подобный показанному ниже. При этом выравнивание по центру изменяется в соответствии со стилем, определенном в таблице стилей CSS.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
         AutoEventWireup="true" CodeBehind="Default.aspx.cs"
         Inherits="EventRegistrationWeb.Default" %>
<asp:Content ID="ContentHead" ContentPlaceHolderID="head" runat="server">
  <style type="text/css">
     .style1
     {
        text-align: center;
     }
  </style>
</asp:Content>
<asp:Content ID="ContentMain" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
  <p class="style1">
    Welcome to the</p>
  <p class="style1">
    Event Registration</p>
  <p class="style1">
    Sample application for Beginning Visual C# 2010!</p>
</asp:Content>
```

Фрагмент кода **Default.aspx**

6. Откройте браузер, чтобы просмотреть страницу. Результат должен выглядеть подобно показанному на рис. 18.20.

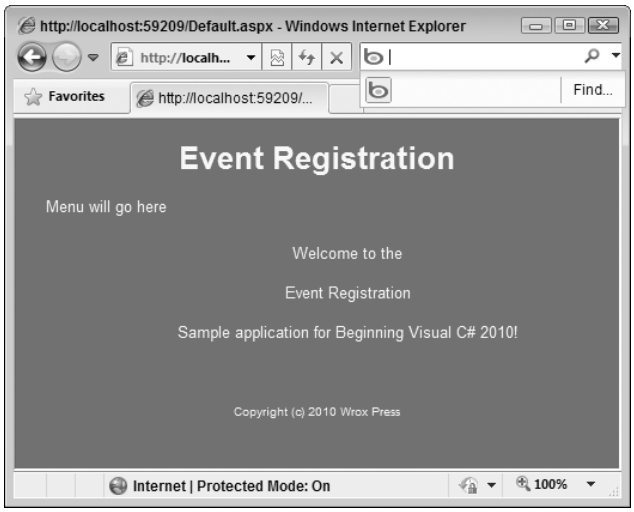


***Рис. 18.20**. Страница, использующая мастер-страницу*

Навигация по сайту

Для обеспечения навигации по множеству страниц веб-сайта можно определить XML-файл, который содержит структуру сайта, и использовать ряд элементов управления для отображения навигационных опций. Наиболее важные элементы управления навигацией перечислены в табл. 18.6.

Таблица 18.6. Элементы управления навигацией

Элемент управления	Описание
`SiteMapDataSource`	Это элемент управления источником данных, который ссылается на любой поставщик данных карты сайта. В панели инструментов Visual Studio этот элемент управления можно найти в разделе Data (Данные)
`Menu`	Элемент управления `Menu` отображает ссылки на страницы, определенные источником данных карты сайта. Элемент управления `Menu` может отображаться горизонтально или вертикально, и он обладает множеством параметров конфигурирования своего стиля
`SiteMapPath`	Элемент управления `SiteMapPath` использует минимальное пространство для отображения текущей позиции страницы в иерархии веб-сайта. При этом можно отображать гиперссылки в виде текста или изображений
`TreeView`	Элемент управления `TreeView` отображает иерархическое представление веб-сайта

В следующем практическом занятии мы добавим к веб-сайту карту сайта и элемент управления меню для осуществления навигации между страницами.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Добавление средств навигации

1. Откройте проект веб-приложения EventRegistrationWeb.
2. Добавьте к веб-сайту новый элемент Site Map (Карта сайта), щелкнув правой кнопкой мыши на проекте в окне Solution Explorer и выбрав в контекстном меню пункт Add New Item (Добавить новый элемент). Оставьте предложенное по умолчанию имя Web.sitemap.
3. Измените содержимое файла, как показано ниже.

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="Default.aspx" title="Home">
    <siteMapNode url="EventRegister.aspx" title="Register"
                 description="Register to an Event" />
    <siteMapNode url="EventList.aspx" title="Event List"
                 description="Lists Events Worldwide" />
    <siteMapNode url="Admin/EventManagement.aspx" title="Event Management"
                 description="Management of Events" roles="Editors" />
  </siteMapNode>
</siteMap>
```

Фрагмент кода ***Web.sitemap***

4. Откройте файл Events.Master.
5. Найдите элемент управления SiteMapDataSource в панели инструментов (на вкладке Data (Данные)) и добавьте его на страницу.
6. Добавьте элемент управления Menu из вкладки Navigation (Навигация) панели инструментов под заголовком “Registration Demo Web” (Демонстрационный веб-сайт регистрации). Установите SiteMapDataSource1 в качестве источника данных.
7. Сконфигурируйте элемент управления Menu, установив его свойство Orientation в Horizontal, свойство StaticDisplayLevels — в 2, а CssClass — в menu.

```
<div class="navigation">
   
  <asp:Menu ID="Menu1" runat="server" DataSourceID="SiteMapDataSource1"
    Orientation="Horizontal" StaticDisplayLevels="2" CssClass="menu">
  </asp:Menu>
  <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</div>
```

Фрагмент кода ***Events.Master***

8. Добавьте следующие правила стилей в файл Site.css, чтобы определить стиль меню:

```
.menu ul li a
{
  background-color: #008085;
  border: 1px #4e667d solid;
  color: #dde4ec;
  display: block;
  line-height: 1.35em;
  padding: 4px 20px;
  text-decoration: none;
  white-space: nowrap;
}
```

```
.menu ul li a:hover
{
  background-color: #bfcbd6;
  color: #465c71;
  text-decoration: none;
}
```

Фрагмент кода ***Site.css***

9. Добавьте элемент управления `SiteMapPath`, расположив его под элементом управления `Menu`.
10. Откройте файл `Default.aspx` в браузере. Обратите внимание на меню и путь, который отражает позицию текущего файла на веб-сайте.
11. С помощью шаблона Web Form using Master Page (Веб-форма, использующая мастер-страницу) создайте новые страницы `EventRegister.aspx` и `EventList.aspx` и выберите мастер-страницу `Events.Master`.
12. Создайте новую папку `Admin`, а в ней — новую страницу `EventManagement.aspx`. Для создания этой страницы снова воспользуйтесь шаблоном Web Form using Master Page.
13. При необходимости можно добавить и другие страницы, указанные в файле `Web.sitemap`, ссылаясь на ту же самую мастер-страницу для отображения нужных пунктов меню.
14. Добавьте в файл `web.config` элемент `siteMap` внутрь элемента `system.web`, как показано ниже:

```
<siteMap defaultProvider="XmlSiteMapProvider" enabled="true">
  <providers>
    <clear />
    <add name="XmlSiteMapProvider"
        description="Default SiteMap Provider"
        type="System.Web.XmlSiteMapProvider"
        siteMapFile="Web.sitemap"
        securityTrimmingEnabled="true" />
  </providers>
</siteMap>
```

Фрагмент кода ***Web.config***

Описание работы

Структура веб-сайта определяется веб-страницами, перечисленными в файле `Web.sitemap`. Этот XML-файл содержит XML-элементы `<siteMapNode>`, размещенные внутри корневого элемента `<siteMap>`. Элемент `<siteMapNode>` определяет веб-страницу. Имя файла страницы устанавливается в атрибуте `url`, а атрибут `title` указывает имя, которое должно отображаться в меню. Иерархия страниц определяется созданием элементов `<siteMapNode>`, дочерних по отношению к странице, где должна размещаться ссылка на дочерние страницы.

Элемент управления `SiteMapDataSource` является элементом управления источником данных. Этот элемент управления может использовать различных поставщиков. По умолчанию для получения доступа к данным применяется класс `XmlSiteMapProvider`, а сам этот класс использует файл `Web.sitemap`.

Поскольку к `siteMapNode` с URL-адресом `EventManagement.aspx` применен атрибут `roles`, только те пользователи, которые перечислены в специальной роли `Editors`, могут видеть эту запись меню. Так как по умолчанию эта функция авторизации `XmlSiteMapProvider` отключена, файл `web.config` изменен для установки свойства

`securityTrimmingEnabled` класса `XmlSiteMapProvider`. Если бы роли не требовались для меню, эта настройка в файле `web.config` вообще была бы не нужной.

Элемент управления `Menu` позволяет редактировать пункты меню, отображаемые в источнике ASPX. Пункты меню можно добавлять также программно. Простейший способ добавления пунктов меню предполагает применение источника данных карты сайта с конфигурированием источника данных.

Аутентификация и авторизация

В целях обеспечения безопасности веб-сайта с помощью *аутентификации* производится проверка того, что данный пользователь имеет действительную учетную запись. С другой стороны, *авторизация* подтверждает, что аутентифицированному пользователю разрешено работать с ресурсом.

В ASP.NET предлагаются два вида аутентификации — Windows и с помощью форм. Для веб-приложений чаще всего применяется аутентификация с помощью форм, которая и рассматривается в настоящем разделе. ASP.NET предоставляет также ряд замечательных новых средств аутентификации с помощью форм. Аутентификация Windows имеет дело с пользовательскими учетными записями Windows и IIS.

В ASP.NET определено множество классов, предназначенных для аутентификации пользователей. Новая архитектура показана на рис. 18.21. ASP.NET делает доступными набор новых элементов управления безопасностью, таких как `Login` или `PasswordRecovery`. Эти элементы управления работают с API-интерфейсом членства (Membership API), который позволяет создавать и удалять пользователей, проверять подлинность регистрационной информации либо извлекать информацию о зарегистрированных в текущий момент пользователях. Сам интерфейс Membership API использует *поставщика членства*. В среде ASP.NET 4 существуют различные поставщики, которые обеспечивают доступ к информации о пользователях в базе данных Access, базе данных SQL Server или Active Directory. Можно также создать нестандартного поставщика, который будет осуществлять доступ к XML-файлу или к любому нестандартному хранилищу.

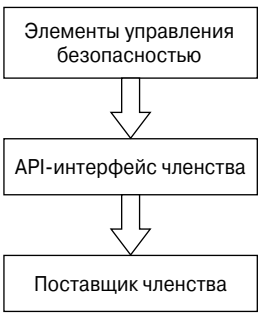


Рис. 18.21. *Архитектура аутентификации пользователей*

Конфигурирование аутентификации

В этой главе рассматривается применение аутентификации с помощью форм и поставщика членства. В следующем практическом занятии мы сконфигурируем настройки безопасности для веб-приложения и назначим разные списки доступа различным папкам.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Конфигурирование настроек безопасности**

1. С помощью Visual Studio откройте ранее созданное веб-приложение `Event RegistrationWeb`.
2. Создайте новую папку `Intro`, выбрав в проводнике решений веб-каталог, а затем пункт меню Website⇨Add Folder⇨Regular Folder (Веб-сайт⇨Добавить папку⇨Обычная папка). Эта папка будет сконфигурирована для доступа всем пользователям, в то время как основная папка будет доступна только аутентифицированным пользователям. Ранее созданная папка `Admin` будет доступна только пользователям, которые перечислены в роли `Editors`.

3. Запустите средство администрирования веб-приложений ASP.NET (ASP.NET Web Application Administration), выбрав в Visual Studio 2010 пункт меню Project⇨ASP.NET Configuration (Проект⇨Конфигурация ASP.NET).
4. Перейдите на вкладку Security (Безопасность), которая показана на рис. 18.22.

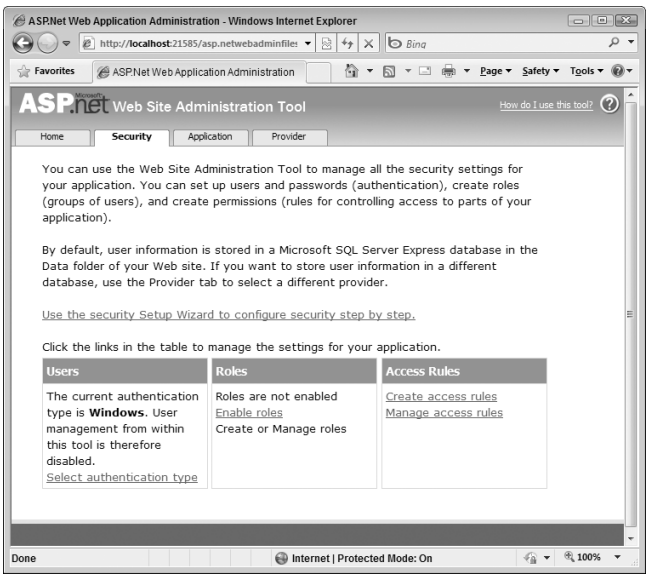


***Рис. 18.22**. Вкладка Security средства администрирования веб-приложений ASP.NET*

5. Щелкните на ссылке Use the security Setup Wizard to configure security step by step (Использовать мастер настройки безопасности для пошагового конфигурирования). В окне приветствия мастера щелкните на кнопке Next (Далее). На втором шаге мастера выберите метод доступа From the internet (Из Интернета), как показано на рис. 18.23.
6. Щелкните на кнопке Next. На третьем шаге отображается сконфигурированный поставщик. По умолчанию это поставщик базы данных SQL Server. Эту настройку нельзя изменять в режиме мастера, но ее можно изменить впоследствии.
7. Щелкните на кнопке Next. В окне Define Roles (Определите роли) установите флажок Enable roles for this Web site (Включить роли для этого веб-сайта).
8. Щелкните на кнопке Next. Создайте новую роль `Editors`.
9. Щелкните на кнопке Next, в результате чего произойдет переход к пятому шагу мастера, где можно будет добавить новых пользователей (рис. 18.24). Создайте две новые учетные записи. Одна из них должна быть членом роли `Editors`.
10. После того как пользователи будут успешно созданы, щелкните на кнопке Next, чтобы перейти к шестому шагу мастера (рис. 18.25). Здесь можно определить то, каким пользователям разрешен либо запрещен доступ к веб-сайту или конкретным каталогам. Добавьте правило запрещения доступа анонимным пользователям. Затем выберите каталог `Intro` и добавьте правило, разрешающее доступ анонимным пользователям. Выберите папку `Admin`, чтобы запретить доступ аутентифицированным пользователям и разрешить доступ пользователям, указанным в роли `Editors`. Затем щелкните на кнопке Next и, наконец, на кнопке Finish (Готово).

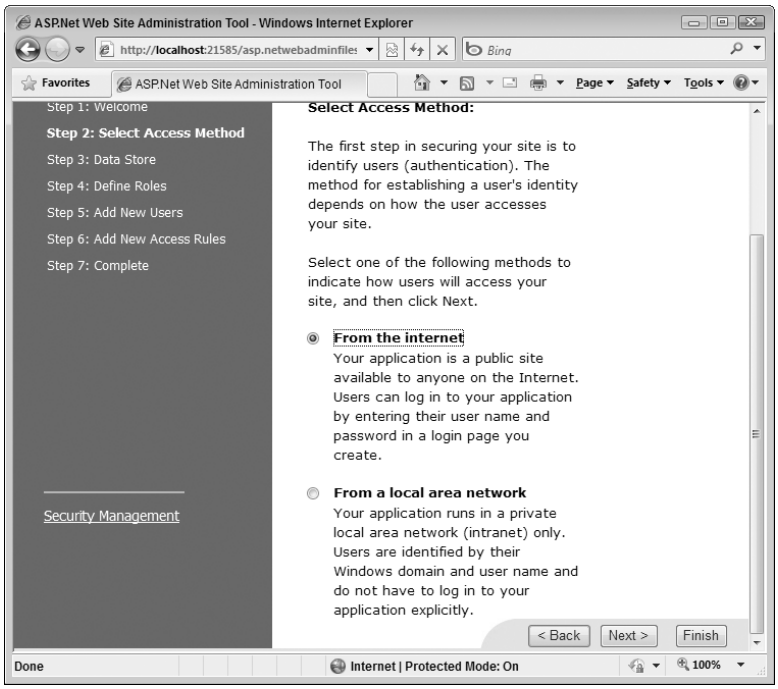


***Рис. 18.23**. Выбор метода доступа*

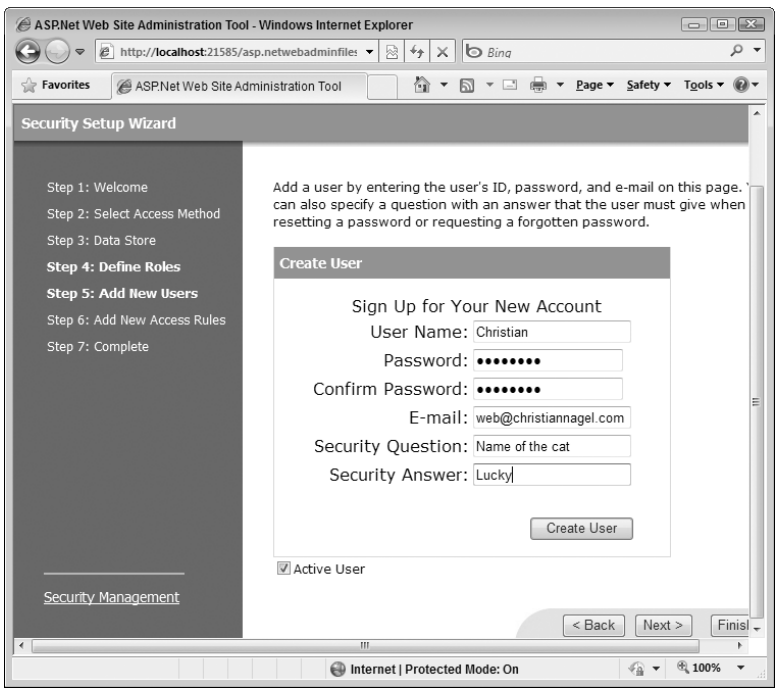


***Рис. 18.24**. Добавление новых пользователей*

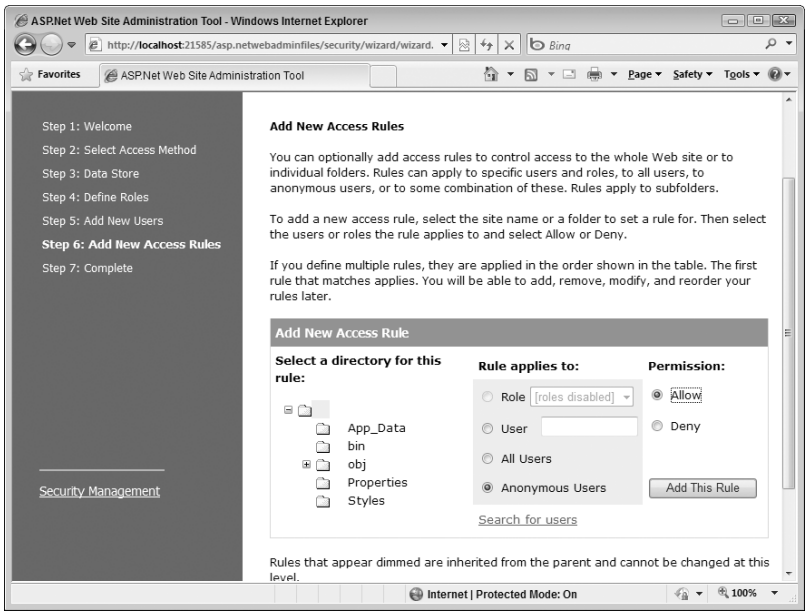


***Рис. 18.25**. Определение правил доступа для пользователей*

Описание работы

По завершении конфигурирования настроек безопасности будет создана новая база данных SQL Server. Обновив файлы в Solution Explorer, вы увидите новую базу данных SQL Server Express `ASP.NETDB.mdf` в каталоге `App_Data`. Эта база данных содержит таблицы, используемые поставщиком членства SQL.

Теперь наряду с веб-приложением будет отображаться и файл конфигурации `web.config`. Этот файл содержит конфигурационные настройки аутентификации с помощью форм, поскольку была выбрана аутентификация через Интернет, и раздел `<authorization>` запрещает доступ анонимным пользователям. При изменении поставщика членства новый поставщик был бы перечислен в файле конфигурации. Поскольку поставщик SQL Server является поставщиком по умолчанию, уже определенным в файле конфигурации компьютера, нет никакой необходимости указывать его здесь:

```
<authorization>
   <deny users="?" />
</authorization>
<roleManager enabled="true" />
<authentication mode="Forms" />
```

Фрагмент кода ***Web.config***

В подпапке `Intro` также присутствует конфигурационный файл — `web.config`. В этом файле раздел аутентификации отсутствует, поскольку информация о конфигурации аутентификации извлекается из родительского каталога. Однако раздел авторизации отличается. В данном случае доступ анонимным пользователям разрешен элементом `<allow users="?" />`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
```

```
        <allow users="?" />
      </authorization>
    </system.web>
</configuration>
```

Фрагмент кода **Intro\Web.config**

В папке Admin присутствует еще один файл конфигурации — web.config. Его раздел авторизации разрешает доступ роли Editors и запрещает доступ аутентифицированным пользователям:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Editors" />
      <deny users="*" />
    </authorization>
  </system.web>
</configuration>
```

Фрагмент кода **Admin\Web.config**

Использование элементов управления безопасностью

В состав ASP.NET входит множество элементов управления безопасностью. Вместо того чтобы создавать нестандартную форму, запрашивающую пользователя имя и пароль, можно воспользоваться готовым элементом управления Login. Элементы управления безопасностью и их функциональность кратко описаны в табл. 18.7.

Таблица 18.7. Элементы управления безопасностью

Элемент управления безопасностью	Описание
Login	Составной элемент управления, который включает в себя элементы управления для запроса имени пользователя и пароля
LoginStatus	Включает гиперссылки для входа или выхода из сайта, в зависимости от того, зарегистрирован пользователь или нет
LoginName	Отображает имя пользователя
LoginView	Позволяет отображать разное содержимое в зависимости от того, зарегистрирован пользователь или нет
PasswordRecovery	Составной элемент управления, предназначенный для восстановления забытого пароля. В зависимости от настроек конфигурации безопасности пользователю предлагается ответить на ранее определенный секретный вопрос, либо пароль отправляется ему по электронной почте
ChangePassword	Составной элемент управления, который позволяет зарегистрированным пользователям изменять свои пароли
CreateUserWizard	Мастер создания нового пользователя и записи информации о пользователе в базу данных поставщика членства

В следующем практическом занятии мы добавим к веб-приложению страницу регистрации.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание страницы регистрации

При попытке запуска веб-сайта после того, как он сконфигурирован для запрещения доступа анонимным пользователям, должно отобразиться сообщение об ошибке, поскольку страница `login.aspx` отсутствует. Если вместе с аутентификацией посредством форм не сконфигурирована специальная страница регистрации, по умолчанию применяется страница `login.aspx`. Давайте создадим эту страницу.

1. Добавьте новый элемент типа Web Form using Master Page (Веб-форма, использующая мастер-страницу) и назначьте ему имя `login.aspx`.
2. Добавьте к форме элемент управления `Login`.
3. Это все, что требуется для создания страницы регистрации. Теперь при запуске сайта `default.aspx` происходит переадресация к странице `login.aspx`, в которой можно ввести регистрационные сведения ранее созданного пользователя.

Описание работы

После добавления элемента управления `Login` в представлении исходного кода отобразится следующий код:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
         AutoEventWireup="true" CodeBehind="Login.aspx.cs"
         Inherits="EventRegistrationWeb.Login" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
  <asp:Login ID="Login1" runat="server">
  </asp:Login>
</asp:Content>
```

Фрагмент кода ***Login.aspx***

Свойства этого элемента управления позволяют конфигурировать текст меток заголовка, имени пользователя, пароля, а также кнопки регистрации. Устанавливая свойство `DisplayRememberMe`, можно делать видимым флажок Remember me next time (Запомнить меня при следующем посещении).

Если требуется большая степень контроля над внешним видом и поведением элемента управления `Login`, его можно преобразовать в шаблон. Это можно выполнить в представлении конструктора, щелкая на смарт-теге и выбирая пункт Convert to Template (Преобразовать в шаблон). Затем при щелчке на Edit Templates (Редактирование шаблонов) откроется представление, в котором можно добавлять и изменять любые элементы управления.

Для проверки полномочий пользователя при щелчке на кнопке Login (Вход) элемент управления вызывает метод `Membership.ValidateUser()`, и это не нужно делать самому.

Когда пользователи не имеют учетной записи для регистрации в веб-сайте `EventRegistration`, они должны создать собственное регистрационное имя. Это легко сделать с помощью элемента управления `CreateUserWizard`, как показано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование мастера CreateUser

1. Добавьте новую веб-страницу `RegisterUser.aspx` в ранее созданной папке `Intro`. Эта папка сконфигурирована для доступа анонимных пользователей.
2. Добавьте на эту веб-страницу элемент управления `CreateUserWizard`.
3. Установите значение свойства `ContinueDestinationPageUrl` в `~/Default.aspx`.

4. Добавьте элемент управления `LinkButton` на страницу `Login.aspx`. В качестве содержимого этого элемента управления установите `Register User`, а в качестве значения его свойства `PostBackUrl` — веб-страницу `Intro/RegisterUser.aspx`.
5. Теперь можно запустить приложение. Щелчок на ссылке Register User (Зарегистрировать пользователя) на странице `Login.aspx` выполняет перенаправление к странице `RegisterUser.aspx`, где и будет создана новая учетная запись с введенными данными.

Описание работы

Элемент управления `CreateUserWizard` — это подобный мастеру элемент управления, который состоит из нескольких шагов мастера, определенных элементом `<WizardSteps>`:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
         AutoEventWireup="true" CodeBehind="RegisterUser.aspx.cs"
         Inherits="EventRegistrationWeb.Intro.RegisterUser" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain" runat="server">
  <asp:CreateUserWizard ID="CreateUserWizard1" runat="server">
    <WizardSteps>
      <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
      </asp:CreateUserWizardStep>
      <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
      </asp:CompleteWizardStep>
    </WizardSteps>
  </asp:CreateUserWizard>
</asp:Content>
```

Фрагмент кода ***RegisterUser.aspx***

Эти шаги мастера могут быть сконфигурированы в представлении конструктора. Смарт-тег элемента управления позволяет конфигурировать каждый из этих шагов отдельно. Конфигурирование шага Sign Up for Your New Account (Выполните подписку для получения новой учетной записи) показано на рис. 18.26. Можно также добавить нестандартные шаги, выполняющие конфигурирование нестандартных элементов управления, чтобы реализовать особые требования, такие как принятие пользователями условий соглашения перед подпиской на учетную запись.

Рис. 18.26*. Конфигурирование шага Sign Up for Your New Account*

Чтение и запись в базе данных SQL Server

Большинству веб-приложений требуется доступ к базе данных для чтения и записи в нее нужной информации. В этом разделе мы создадим новую базу данных для хранения информации о событиях и научимся использовать ее из среды ASP.NET. Вначале в следующем практическом занятии мы создадим новую базу данных SQL Server. Это можно сделать непосредственно в Visual Studio 2010.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание новой базы данных

1. Откройте ранее созданное веб-приложение `EventRegistrationWeb`.
2. Откройте окно Server Explorer (Проводник сервера), выбрав пункт меню View⇨Other Windows⇨Server Explorer (Вид⇨Другие окна⇨Проводник сервера).
3. В окне Server Explorer выберите опцию Data Connections (Подключения к данным), щелкните правой кнопкой мыши, чтобы открыть контекстное меню, и выберите пункт Create New SQL Server Database (Создать новую базу данных SQL Server). Откроется диалоговое окно Create New SQL Server Database (Создание новой базы данных SQL Server), показанное на рис. 18.27.
4. Введите **`(local)\sqlexpress`** в качестве имени сервера и **`BegVCSharpEvents`** для имени базы данных.

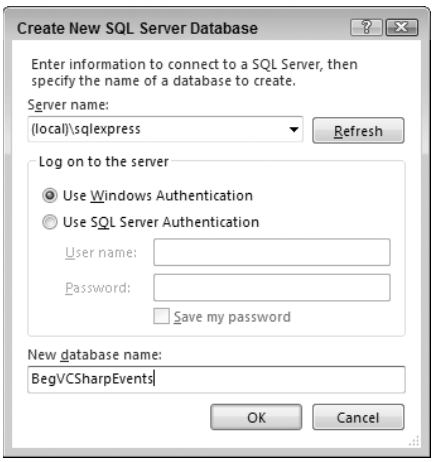


Рис. 18.27. *Диалоговое окно* Create New SQL Server Database

5. После того как база данных будет создана, выберите новую базу данных в проводнике сервера.
6. Выберите запись Tables (Таблицы) в структуре базы данных и выберите пункт меню Data⇨Add New⇨Table (Данные⇨Добавить новую...⇨Таблицу) в Visual Studio.
7. Введите следующие имена и типы данных столбцов:

Имя столбца	Тип данных
`Id`	`int`
`Title`	`nvarchar(50)`
`Date`	`datetime`
`Location`	`nvarchar(50)`

8. Сконфигурируйте столбец `ID` в качестве столбца первичного ключа с приращением `1` и начальным значением `1`. Сконфигурируйте все столбцы, запретив для них нулевые значения.
9. Сохраните таблицу под именем `Events`.
10. Добавьте в таблицу несколько событий с какими-то заголовками, датами и местоположениям.

Панель инструментов содержит отдельный раздел Data (Данные), представляющий элементы управления данными, которые позволяют отображать и редактировать данные. Элементы управления данными можно разделить на две категории: представление данных и источник данных. Элемент управления источником данных связан с источником данных, таким как XML-файл, база данных SQL или класс .NET. Представления данных подключаются к источнику данных для представления данных. Все элементы управления данными описаны в табл. 18.8.

Таблица 18.8. Элементы управления данными

Элемент управления данными	Описание
`GridView`	Отображает данные в виде строк и столбцов
`DataList`	Отображает единственный столбец для вывода всех элементов
`DetailsView`	При наличии внутри данных отношения "главные/детали" этот элемент управления можно использовать совместно с элементом управления `GridView`
`FormView`	Отображает одиночную строку источника данных
`Repeater`	Этот элемент управления создается на основе шаблона. С его помощью можно определить HTML-элементы, которые должны генерироваться на основе данных, полученных из источника
`ListView`	Этот элемент управления на основе шаблона подобен элементу управления `Repeater`

Элементы управления источниками данных и их функциональность кратко описаны в табл. 18.9.

Таблица 18.9. Элементы управления источниками данных

Элемент управления источником данных	Описание
`SqlDataSource`	Осуществляет доступ к SQL Server или любому другому поставщику данных ADO.NET (например, Oracle, ODBC и OLE DB). Внутренне этот элемент управления использует класс `DataSet` или `DataReader`
`AccessDataSource`	Позволяет использовать базу данных Access
`EntityDataSource`	Этот элемент управления появился в .NET 4.0. Он позволяет использовать в качестве источника данных ADO.NET Entity Framework.
`ObjectDataSource`	Позволяет использовать в качестве источника данных классы .NET
`XmlDataSource`	Позволяет получать доступ к XML-файлам. При использовании этого источника данных можно отображать иерархические структуры
`SiteMapDataSource`	Использует XML-файлы для определения структуры сайта для создания ссылок внутри веб-сайта. Это средство рассматривается в главе 20

В следующем практическом занятии мы используем элемент управления `GridView` для отображения и редактирования данных из ранее созданной базы данных.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование элемента управления `GridView` для отображения данных

1. Откройте ранее созданную веб-страницу `EventsManagement.aspx` из папки `Admin`.
2. Добавьте в веб-страницу элемент управления `GridView`.
3. В поле со списком Choose Data Source (Выберите источник данных) смарт-тега элемента управления выберите элемент <New data source> (<Новый источник данных>). Откроется диалоговое окно Data Source Configuration Wizard (Мастер конфигурирования источников данных), показанное на рис. 18.28.

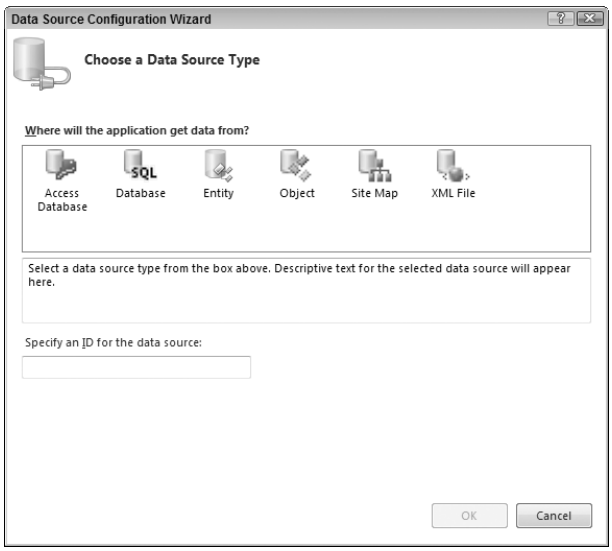


Рис. 18.28. *Диалоговое окно* Data Source Configuration Wizard

4. Выберите элемент Database (База данных) и введите `EventsDataSource` в качестве имени для нового источника данных.
5. Щелкните на кнопке OK, чтобы сконфигурировать источник данных. Откроется диалоговое окно Configure Data Source (Конфигурирование источника данных). Щелкните на кнопке New Connection (Новое подключение), чтобы создать новое подключение.
6. В диалоговом окне Add Connection (Добавление подключения) в качестве имени сервера введите `(local)\sqlexpress` и выберите ранее созданную базу данных `BegVCSharpEvents`. Щелкните на кнопке Test Connection (Проверить подключение), чтобы убедиться в корректности конфигурирования подключения. Если все в порядке, щелкните на кнопке OK. Откроется следующее диалоговое окно (рис. 18.29) для сохранения строки подключения.
7. Отметьте флажок, чтобы сохранить подключение, и введите имя строки подключения — `EventsConnectionString`. Щелкните на кнопке Next.
8. В следующем диалоговом окне для чтения данных выберите таблицу `Events`, как показано на рис. 18.30. Чтобы определить SQL-команду на этом рисунке, выберите столбцы `ID`, `Title`, `Date` и `Location`. Щелкните на кнопке Next.

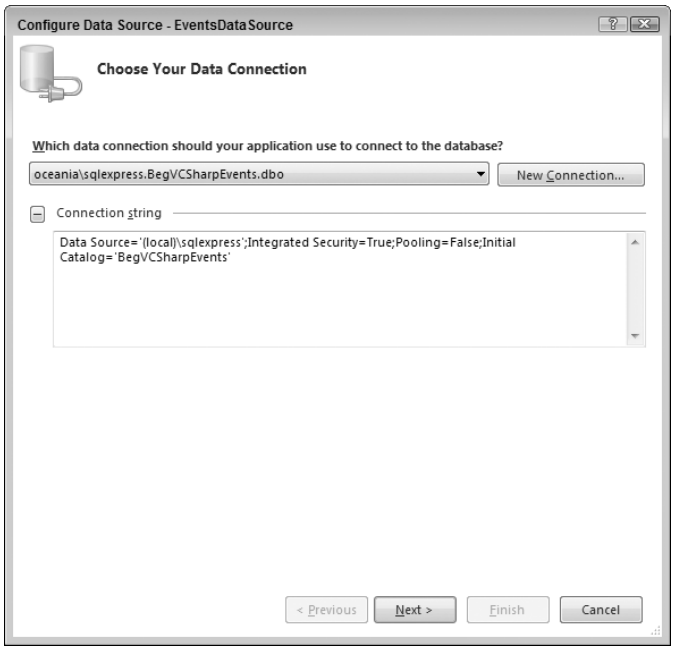


***Рис. 18.29**. Сохранение строки подключения*

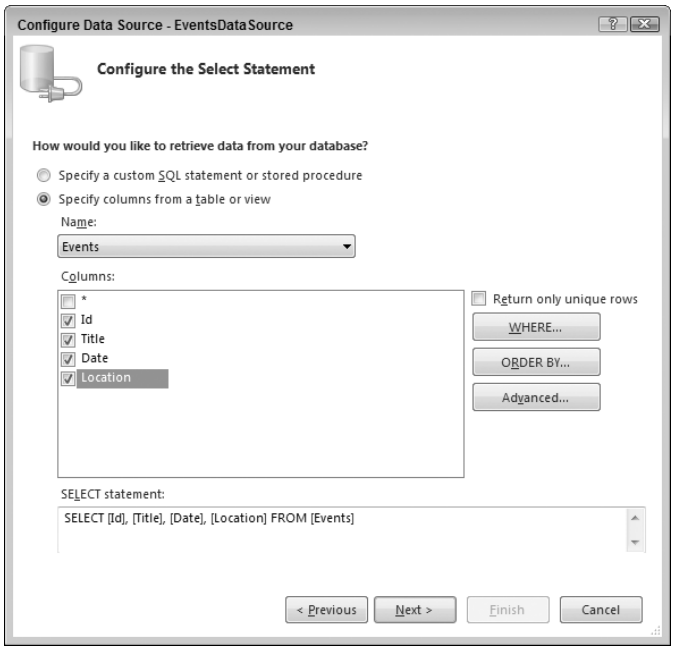


***Рис. 18.30**. Настройка оператора чтения из базы данных*

9. Последний экран диалогового окна Configure Data Source позволяет проверить запрос. По завершении щелкните на кнопке Finish (Готово).
10. Теперь в представлении конструктора отображается элемент управления `GridView` с фиктивными данными и элемент управления `SqlDataSource` по имени `EventsDatasource`.
11. Чтобы придать элементу `GridView` более привлекательный внешний вид, выберите опцию AutoFormat смарт-тега и схему Mocha, как показано на рис. 18.31.
12. При запуске страницы в среде Visual Studio события отобразятся в аккуратной таблице, похожей на показанную на рис. 18.32.

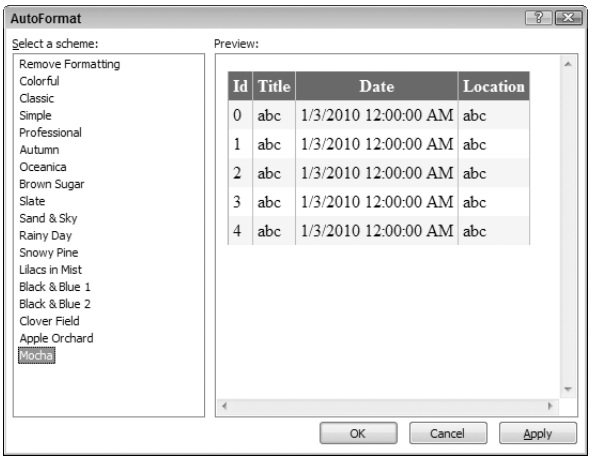


***Рис. 18.31**. Установка параметров автоформатирования*

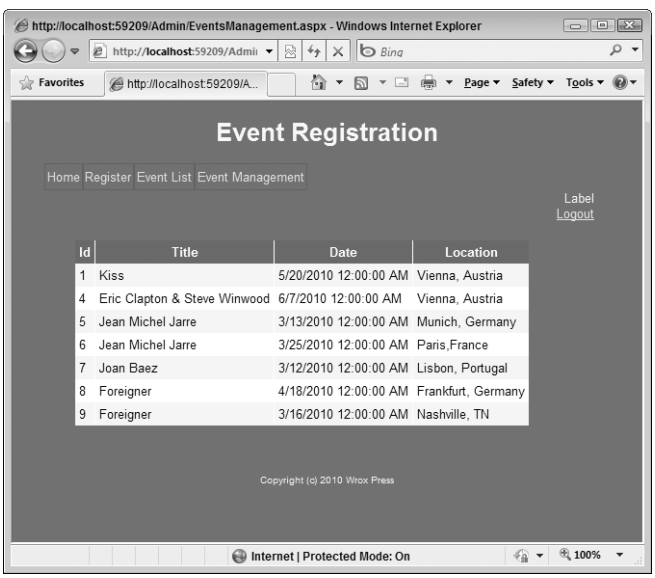


***Рис. 18.32**. Отображение данных с использованием элемента управления `GridView`*

Описание работы

После добавления элемента управления GridView его конфигурация появляется в исходном коде. Атрибут DataSourceID определяет связь с элементом управления источником данных, который расположен за элементом управления сеткой. Внутри элемента <Columns> отображаются все связанные столбцы, необходимые для отображения данных. HeaderText определяет текст заголовка, а DataField — имя поля внутри источника данных.

Источник данных определяется элементом <asp:SqlDataSource>, при этом SelectCommand указывает способ чтения данных из базы, а ConnectionString — способ подключения к базе данных. Поскольку был выбран вариант сохранения строки подключения в файле конфигурации, <%$ используется для создания связи с динамически сгенерированным классом из файла конфигурации.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Events.Master"
          AutoEventWireup="true" CodeBehind="EventsManagement.aspx.cs"
          Inherits="EventRegistrationWeb.Admin.EventsManagement" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolderMain"
             runat="server">
  <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
                BackColor="White" BorderColor="#DEDFDE" BorderStyle="None"
                BorderWidth="1px" CellPadding="4" DataKeyNames="Id"
                DataSourceID="EventsDataSource" ForeColor="Black"
                GridLines="Vertical" PageSize="5">
    <AlternatingRowStyle BackColor="White" />
    <Columns>
     <asp:BoundField DataField="Id" HeaderText="Id" InsertVisible="False"
          ReadOnly="True" SortExpression="Id" />
     <asp:BoundField DataField="Title" HeaderText="Title"
          SortExpression="Title" />
     <asp:BoundField DataField="Date" HeaderText="Date" SortExpression="Date" />
     <asp:BoundField DataField="Location" HeaderText="Location"
          SortExpression="Location" />
    </Columns>
    <FooterStyle BackColor="#CCCC99" />
    <HeaderStyle BackColor="#6B696B" Font-Bold="True" ForeColor="White" />
    <PagerStyle BackColor="#F7F7DE" ForeColor="Black" HorizontalAlign="Right" />
    <RowStyle BackColor="#F7F7DE" />
    <SelectedRowStyle BackColor="#CE5D5A" Font-Bold="True" ForeColor="White" />
    <SortedAscendingCellStyle BackColor="#FBFBF2" />
    <SortedAscendingHeaderStyle BackColor="#848384" />
    <SortedDescendingCellStyle BackColor="#EAEAD3" />
    <SortedDescendingHeaderStyle BackColor="#575357" />
  </asp:GridView>
<asp:SqlDataSource ID="EventsDataSource" runat="server"
  ConnectionString="<%$ ConnectionStrings:BegVCSharpEventsConnectionString %>"
  SelectCommand="SELECT [Id], [Title], [Date], [Location] FROM [Events]">
</asp:SqlDataSource>
</asp:Content>
```

Фрагмент кода ***EventsManagement.aspx***

Файл конфигурации web.config содержит также строку подключения к базе данных:

```
<connectionStrings>
  <add name="BegVCSharpEventsConnectionString"
    connectionString="Data Source='(local)\sqlexpress';
    Integrated Security=True;Pooling=False;
    Initial Catalog='BegVCSharpEvents'"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Фрагмент кода ***web.config***

Теперь нужно по-другому сконфигурировать элемент управления `GridView`. В следующем практическом занятии будет сделано так, чтобы идентификаторы больше не отображались пользователю, а элемент даты-времени выводил только дату.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Конфигурирование элемента управления `GridView`

1. Выберите смарт-тег элемента управления `GridView`, а затем пункт меню Edit Columns (Редактировать столбцы). Откроется диалоговое окно Fields (Поля), показанное на рис. 18.33. Выберите поле `Id` и измените значение его свойства `Visible` на `False`. Это диалоговое окно позволяет организовывать столбцы, изменять цвета и определять текст заголовка. Установите свойство `DataFormatString` столбца `Date` в `{0:D}`, чтобы он отображал только дату без времени.

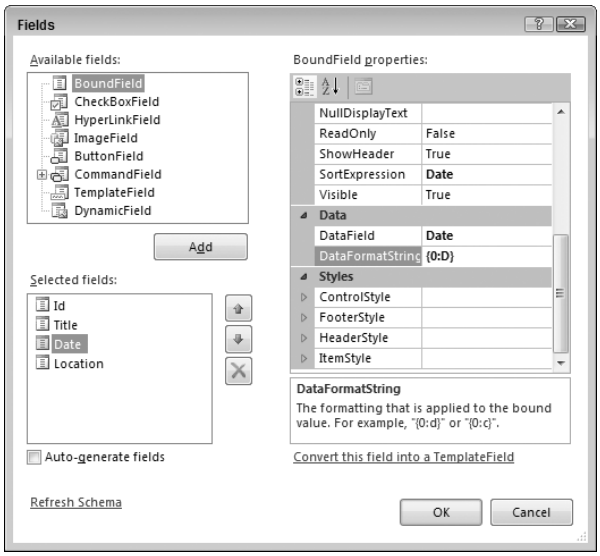


***Рис. 18.33**. Диалоговое окно Fields*

2. Чтобы можно было выполнять редактирование элемента управления `GridView`, для источника данных должна быть определена команда обновления. Выберите элемент управления `SqlDataSource` по имени `EventsDataSource`, а затем выберите опцию Configure Data Source (Конфигурировать источник данных) смарт-тега. В диалоговом окне Configure Data Source (Конфигурирование источника данных) щелкайте на кнопке Next (Далее) до тех пор, пока не отобразится ранее сконфигурированная команда `SELECT`. Щелкните на кнопке Advanced (Дополнительно) и отметьте флажок Generate INSERT, UPDATE, and DELETE statements (Генерировать операторы `INSERT`, `UPDATE` и `DELETE`), как показано на рис. 18.34. Щелкните на кнопке OK. Затем щелкните на кнопках Next и Finish (Готово).
3. Снова выберите смарт-тег элемента управления `GridView`. Теперь меню этого смарт-тега содержит пункт Enable Editing (Разрешить редактирование). После отметки флажка разрешения редактирования к элементу управления `GridView` добавляется новый столбец. С помощью меню смарт-тега можно также редактировать столбцы для размещения новой кнопки Edit (Редактировать) нужным образом. В дополнение к выполненному выберите опции Enable Paging (Разрешить разбиение на страницы), Enable Sorting (Разрешить сортировку) и Enable Selection (Разрешить выбор).

4. Запустите приложение и попробуйте отредактировать существующие записи событий. Чтобы выполнить сортировку, щелкните на заголовке.

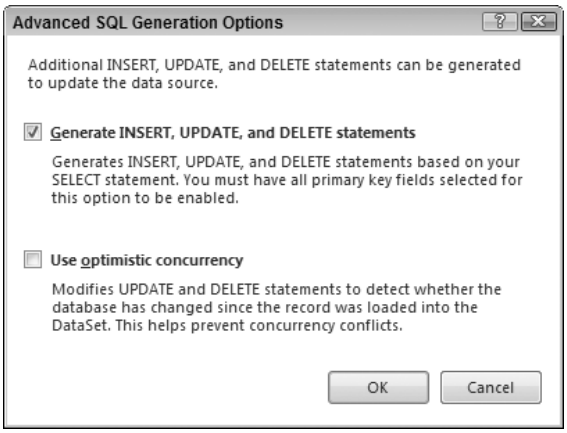


***Рис. 18.34**. Включение генерации операторов `INSERT`, `UPDATE` и `DELETE` для источника данных*

Описание работы

В этом примере не пришлось писать вручную ни одной строки кода. Все было выполнено с помощью веб-элементов управления ASP.NET. Но "за кулисами" эти элементы управления используют многие функции.

Например, элемент управления `SqlDataSource` с помощью `SqlDataAdapter` заполняет `DataSet` информацией из базы данных. Данные, используемые для заполнения `DataSet`, определяются строкой подключения и командой `SELECT`. Простым изменением свойства элемента управления `SqlDataSource` вместо `DataSet` можно применять `SqlDataReader`. Кроме того, установка свойства `EnableCaching` в `true` приводит к автоматическому использованию объекта `Cache` (который рассматривался ранее в этой главе).

Резюме

В этой главе была описана архитектура ASP.NET, работа с элементами управления серверной стороны и некоторые основные функциональные возможности ASP.NET. Как было показано на примерах элементов управления регистрацией и данными, ASP.NET предоставляет ряд элементов управления, не требующих большого объема кодирования.

После ознакомления с основными особенностями работы ASP.NET с серверными элементами управления и механизмом обработки событий была рассмотрена проверка достоверности ввода, а также несколько методов управления состоянием, аутентификацией и авторизацией. Кроме того, было показано, как отображать информацию из базы данных.

Следующие упражнения помогут расширить возможности веб-приложения, разработанного в этой главе.

Упражнения

1. Добавьте имя пользователя к созданной в этой главе мастер-странице. Для этого можно использовать элемент управления `LoginName`. Применяйте элемент управления `LoginView` для отображения этой информации только в том случае, если пользователь аутентифицирован.

2. Измените источник данных страницы `Registration.aspx`, чтобы для отображения событий она использовала базу данных `Events`.
3. Создайте новый проект типа ASP.NET Web Application (Веб-приложение ASP.NET). Просмотрите все файлы и папки, созданные на основе этого шаблона проекта. Все они должны выглядеть знакомо.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Использование серверных веб-элементов управления	Серверные веб-элементы управления — это элементы управления серверной стороны, которые генерируют HTML-разметку. Эти элементы используются аналогично элементам управления Windows
Использование обратных отправок ASP.NET	Модель обратной отправки ASP.NET — очень важная концепция при создании веб-приложений ASP.NET. Код серверной стороны вступает в действие только при обратных отправках серверу. Теперь ASP.NET Ajax позволяет также определять обратные отправки Ajax, при которых обновляются только отдельные части страниц
Верификация пользовательского ввода с помощью элементов управления проверкой достоверности	ASP.NET предоставляет несколько элементов управления проверкой достоверности, которые можно легко использовать для верификации пользовательского ввода как на клиентской, так и на серверной стороне. Проверка на клиентской стороне осуществляется по соображениям производительности, но поскольку веб-клиенту никогда нельзя полностью доверять, проверка должна быть реализована также и на серверной стороне
Управление состоянием	При создании веб-приложений необходимо продумывать место хранения состояния. На стороне клиента это может быть реализовано в виде cookie-наборов или состояния представления. На стороне сервера используются объекты сеанса, кэша и приложения
Мастер-страницы	Мастер-страницы служат для выделения общих частей нескольких страниц в шаблон
Навигация	Для навигации между различными страницами веб-сайта можно применять элементы управления меню. Вместо того чтобы добавлять ссылки на страницы непосредственно в элемент управления меню, меню можно связать с картой сайта
Чтение и запись в базу данных SQL Server	Доступ к базе данных абстрагируется с помощью элементов управления ASP.NET. Элемент управления `GridView` можно легко настраивать индивидуально из представления визуального конструктора. Информация для этой сетки может быть получена из источника данных, в котором для обеспечения чтения и записи из базы данных достаточно установить соответствующие свойства (вместо написания кода C#)

19

Веб-службы

В ЭТОЙ ГЛАВЕ...

- Обзор веб-служб
- Создание веб-служб с помощью ASP.NET
- Использование веб-службы из приложения Windows Forms
- Использование веб-службы из клиента ASP.NET
- Асинхронный вызов веб-служб
- Передача данных между веб-службами

В то время как веб-приложения служат пользователю интерфейсом для получения доступа к функциональным возможностями приложения, веб-службы являются интерфейсом приложений для получения доступа к функциональности приложения. Веб-службы — это программы серверной стороны, которые прослушивают сообщения, поступающие от клиентских приложений, и возвращают конкретную информацию. Эта информация может поступать от самой веб-службы, от других компонентов в данном домене или от других веб-служб.

В этой главе не рассматриваются подробно нюансы внутренней работы веб-служб, но будет предоставлено достаточно информации, чтобы с помощью Visual Studio приступить к созданию и использованию простых веб-служб ASP.NET.

Использование веб-служб

Чтобы взглянуть на веб-службы под еще одним углом, можно сделать различие между обменом данными *между пользователем и приложением* и *между приложениями*. Начнем с рассмотрения примера обмена данными между пользователем и приложением: получение информации о погоде из Интернета. Несколько веб-сайтов, таких как `weather.msn.com` и `www.weather.com`, предоставляют сведения о погоде в легко систематизируемом формате. Обычно пользователь читает эти страницы непосредственно.

Если же требовалось создать многофункциональное клиентское приложение для отображения информации о погоде (выполняющее обмен данными между приложениями), приложение должно было бы подключаться к веб-сайту посредством строки URL-адреса, содержащей название интересующего города. Результирующее HTML-сообщение, возвращенное веб-сайтом, нужно было бы проанализировать для извлечения информации о температуре и погодных условиях, после чего информацию, наконец, можно было бы отобразить в соответствующем формате клиентского приложения.

Эта чрезвычайно трудоемкая задача, учитывая, что требуется всего лишь получить набор температурных показателей для конкретного города, а процесс извлечения данных из HTML-сообщения не прост, поскольку эти данные предназначены для отображения в веб-браузере и не предусматривают их использования каким-либо другим клиентским бизнес-приложением. Поэтому данные внедрены в текст и не доступны для простого извлечения. В результате для получения с этой же веб-страницы других данных (вроде сведений об осадках) клиентское приложение пришлось бы переписывать или адаптировать. В отличие от веб-браузера, приложение позволяет пользователям немедленно выбирать нужные данные и пропускать ненужные.

Для решения проблемы обработки HTML-данных веб-служба предлагает средства возврата только запрошенных данных. Для этого достаточно вызвать метод на удаленном сервере и получить необходимую информацию, которая может непосредственно обрабатываться клиентским приложением. К счастью, в этом случае приходится иметь дело с предварительно сформатированным текстом, предназначенным для интерфейса пользователя, поскольку веб-служба представляет информацию в формате XML, а для обработки XML-данных уже существует набор инструментальных средств. Единственное, что требуется от клиентского приложения — вызов ряда методов XML-классов .NET Framework для получения информации. Более того, при создании клиентского приложения для веб-службы .NET на языке C# для этого даже не требуется написание кода — существуют средства, которые сгенерируют код C# автоматически.

Подобное приложение вывода информации о погоде — один из примеров множества возможных применений веб-служб.

Сценарий использования приложения бюро путешествий

Как вы планируете проведение своего отпуска? Вместо того чтобы предоставить все планирование бюро путешествий, свои выходные можно распланировать через Интернет. На веб-сайте авиакомпании можно просмотреть имеющиеся рейсы и заказать необходимые билеты. С помощью поисковых механизмов можно найти отель в нужном городе. Во многих случаях можно также найти карту с указанием маршрута до отеля. Обнаружив домашнюю страницу отеля, следует перейти на страницу формы заказа и забронировать номер. Затем можно поискать фирму по прокату автомобилей и т.д.

На сегодня поиск подходящих веб-сайтов с помощью поисковых механизмов и последующий переход к этим сайтам сопряжен с огромным объемом работы. Вместо того чтобы проделывать все это, можно было бы создать приложение "Домашнее бюро путешествий", использующее веб-службы со сведениями об отелях, авиалиниях, фирмах по прокату автомобилей и т.п. Затем клиенту можно было бы предложить простой интерфейс, учитывающий все аспекты отпуска, включая заблаговременный заказ билетов на конкретный музыкальный концерт. С помощью своего мобильного устройства во время отпуска эти же веб-службы можно применять для получения карты расположения определенных мест отдыха и точной информации о культурных событиях, киносеансах и т.п.

Сценарий использования приложения распространения книг

Веб-службы могут также быть полезны для двух компаний, поддерживающих партнерские отношения. Предположим, что распространитель книг желает развернуть книжные магазины с информацией об имеющихся в наличии книгах. Эту задачу можно выполнить с помощью веб-службы. Можно создать приложение ASP.NET, использующее веб-службу, которое будет предоставлять эту услугу непосредственно пользователям. Еще одним клиентским приложением этой службы является Windows-приложение книжного магазина, которое вначале проверяет наличие указанных книг на локальном складе, а затем на складе дистрибьютора. Продавец может немедленно ответить на вопросы о сроках доставки, не обращаясь к другим источникам складской информации в других приложениях.

Типы клиентских приложений

Клиентом веб-службы может быть многофункциональное Windows-приложение, созданное с помощью Windows Forms, либо приложение WPF, Silverlight или ASP.NET, использующее Web Forms. Для потребления веб-службы может применяться ПК под управлением Windows, система UNIX или мобильное устройство. Платформа .NET Framework позволяет применять веб-службы в любом типе приложений.

Архитектура приложения

Как же в действительности выглядит приложение, использующее веб-службы? Вызов веб-служб аналогичен для приложений ASP.NET, Windows или приложений для портативных устройств (для описанных выше сценариев). Веб-службы являются важной технологией для всех этих типов приложений.

Сценарий возможной работы с веб-службами показан на рис. 19.1. Устройства и браузеры подключаются к приложению ASP.NET, разработанному с помощью Web Forms, через Интернет. Это приложение ASP.NET использует локальные и удаленные, доступные по сети, веб-службы: портальные веб-службы, веб-службы, специфичные для приложения, и унифицированные элементарные веб-службы.

Следующий перечень должен помочь в понимании смысла этих типов служб.

- **Портальные веб-службы.** Предоставляют службы различных компаний по одной тематике. Обеспечивают простую точку доступа к нескольким службам.
- **Веб-службы, специфичные для приложения.** Созданы для применения отдельным приложением.
- **Унифицированные элементарные веб-службы.** Службы, которые легко использовать внутри нескольких приложений.

Показанные на рис. 19.1 Windows-приложения могут использовать веб-службы непосредственно, не обращаясь к приложению ASP.NET.

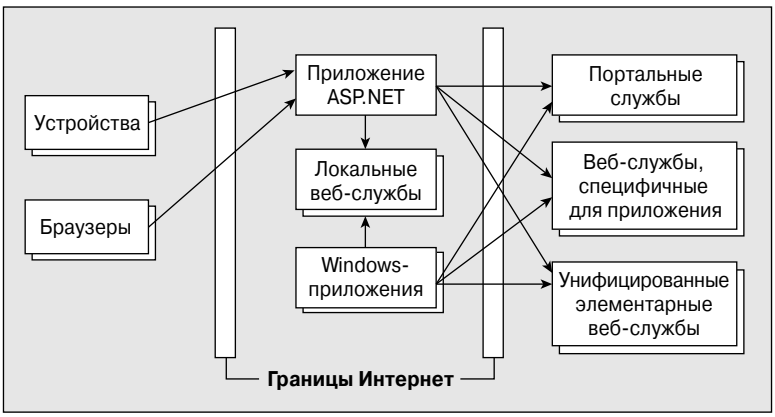


Рис. 19.1. *Использование веб-служб*

Архитектура веб-служб

Веб-службы могут использовать протокол SOAP, являющийся стандартом, который определен многими компаниями. Огромное преимущество веб-служб — их независимость от платформы. Однако веб-службы — полезная технология не только, когда необходимо обеспечить взаимодействие нескольких платформ. Они могут применяться при разработке приложений .NET как для клиентской, так и для серверной стороны. Преимущество этой технологии в том, что клиент и сервер могут быть независимыми друг от друга. Описание службы создается документом WSDL (Web Service Description Language — язык описания веб-служб), который может быть разработан независимо от новых версий веб-службы и, следовательно, клиент не будет нуждаться в изменении.

Ниже более подробно рассматриваются необходимые шаги.

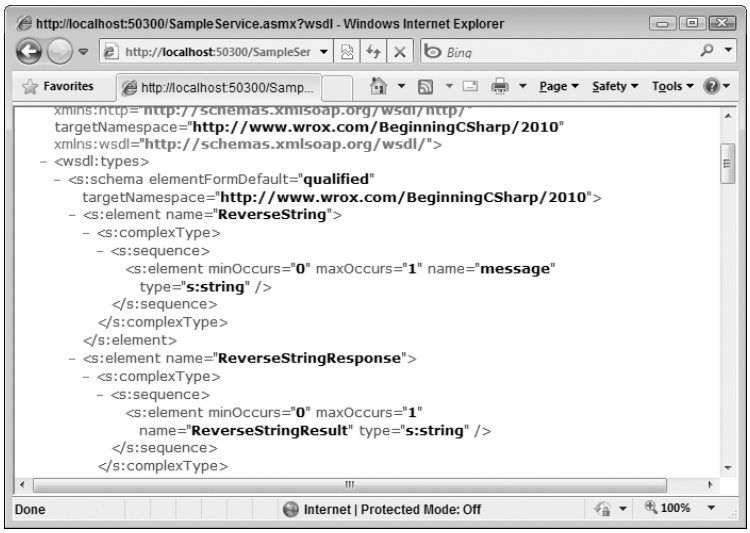
Вызов методов и язык описания веб-служб

Документ WSDL содержит информацию о методах, поддерживаемых веб-службой, способах их вызова, а также о типах параметров, переданных службе и возвращенных службой. Документ WSDL, генерируемый исполняющей средой ASP.NET, показан на рис. 19.2. Добавление строки `?wsdl` в файл `.asmx` приводит к возврату документа WSDL.

Обработку этой информации не обязательно осуществлять непосредственно. Атрибут `WebMethod` (рассматривается позднее) будет приводить к динамической генерации документа WSDL. Добавление веб-ссылки в клиентское приложение средствами Visual Studio вызывает запрос документа WSDL. Этот документ WSDL, в свою очередь, применяется для создания прокси-клиента с теми же методами и аргументами. Использование этого прокси-

клиента обеспечивает приложению то преимущество, что ему приходится только вызывать методы, как они реализованы на сервере, поскольку для выполнения вызова по сети прокси-клиент преобразует их в вызовы SOAP.

Спецификация WSDL поддерживается консорциумом World Wide Web Consortium (W3C). С ней можно ознакомиться на веб-сайте W3C по адресу `www.w3.org/TR/wsdl`.




```
targetNamespace="http://www.wrox.com/BeginningCSharp/2010"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
- <wsdl:types>
  - <s:schema elementFormDefault="qualified"
      targetNamespace="http://www.wrox.com/BeginningCSharp/2010">
    - <s:element name="ReverseString">
      - <s:complexType>
        - <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="message"
              type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    - <s:element name="ReverseStringResponse">
      - <s:complexType>
        - <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
              name="ReverseStringResult" type="s:string" />
          </s:sequence>
        </s:complexType>
```

Рис. 19.2. *Пример документа WSDL*

Вызов метода

Сообщение SOAP — основной элемент обмена данными между клиентом и сервером. Чтобы метод можно было вызвать применительно к веб-службе, вызов должен быть преобразован в сообщение SOAP, как это определено в документе WSDL. Части сообщения SOAP показаны на рис. 19.3. *Конверт* SOAP, как легко догадаться, заключает всю информацию SOAP в единый блок. Сам конверт SOAP состоит из двух частей: *заголовка* SOAP и *тела* SOAP. Необязательный заголовок определяет то, как клиент и сервер должны обрабатывать тело сообщения. Обязательное тело SOAP содержит пересылаемые данные. Обычно информацией внутри тела является вызываемый метод и упорядоченные значения параметров. Сервер SOAP отсылает обратно возвращаемые значения в теле SOAP сообщения SOAP.

Рис. 19.3. *Сообщение SOAP*

В следующем примере показано, как выглядит сообщение SOAP, отправленное клиентом серверу. Клиент вызывает метод `ReverseString()` веб-службы. Строка `Hello World!` передается этому методу в качестве аргумента. Как видите, вызов метода расположен внутри тела SOAP, в XML-элементе `<soap:Body>`. Само тело содержится внутри конверта `<soap:Envelope>`. Перед сообщением SOAP расположен заголовок HTTP, поскольку сообщение SOAP отправляется с помощью HTTP-запроса `POST`.

Создание такого сообщения не обязательно, поскольку это делает прокси-клиент.

```
POST /Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 508
SOAPAction: "http://www.wrox.com/webservices/ReverseString"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseString xmlns="http://www.wrox.com/webservices">
      <message>Hello World!</message>
    </ReverseString>
  </soap:Body>
</soap:Envelope>
```

Как видно из XML-элемента `ReverseStringResult`, сервер отвечает SOAP-сообщением `!dlroW olleH`:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 446
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ReverseStringResponse xmlns="http://www.wrox.com/webservices">
      <ReverseStringResult>!dlroW olleH</ReverseStringResult>
    </ReverseStringResponse>
  </soap:Body>
</soap:Envelope>
```

Поддержкой спецификации SOAP занимается рабочая группа протокола XML консорциума W3C (`www.w3.org/TR/soap`).

Базовый профиль WS-I

Спецификация SOAP и другие созданные на ее основе спецификации сформировались не сразу. В результате появились изменения и множество различных версий, которые затрудняют взаимодействие между службами различных поставщиков. Для решения этой проблемы была создана Организация по обеспечению взаимодействия веб-служб (Web Services Interoperability Organization). Эта организация определяет требования к веб-службам в своих спецификациях WS-I Basic Profile (Базовый профиль взаимодействия веб-служб). Со спецификацией WS-I Basic Profile можно ознакомиться на сайте `http://www.ws-i.org`. Веб-службы, разработанные с помощью ASP.NET, соответствуют спецификации Basic Profile 1.1, которая определена в документе по адресу `http://www.ws-i.org/Profiles/BasicProfile-1.1.html`.

Веб-службы и .NET Framework

Платформа .NET Framework позволяет легко создавать и применять веб-службы, используя три основных пространства имен, связанные с веб-службами:

- `System.Web.Services` — классы этого пространства имен служат для создания веб-служб;
- `System.Web.Services.Description` — позволяет описывать веб-службы посредством WSDL;
- `System.Web.Services.Protocols` — позволяет создавать запросы и ответы SOAP.

Веб-службы можно создавать с помощью ASP.NET или Windows Communication Foundation (WCF). Технология WCF обладает значительно большей гибкостью, чем веб-службы ASP.NET, поскольку предоставляет различные возможности размещения веб-служб, не привязана к ASP.NET и поддерживает различные протоколы, а не только HTTP. Преимущество веб-служб ASP.NET заключается в том, что они несколько проще в использовании. Кроме того, шаблоны WCF не были доступны в версии Visual Studio 2008 Express. Технология WCF рассматривается в главе 27.

Создание веб-службы

Чтобы реализовать веб-службу, необходимо создать класс веб-службы, производный от `System.Web.Services.WebService`. Класс `WebService` предоставляет доступ к объектам `Application` и `Session` из ASP.NET. Использовать этот класс не обязательно, а создавать производный от него класс нужно, только если требуется простой доступ к предоставляемым им свойствам.

Свойства класса `WebService` описаны в табл. 19.1.

Таблица 19.1. Свойства класса `WebService`

Свойство	Описание
`Application`	Возвращает объект `HttpApplicationState` для текущего запроса
`Context`	Возвращает объект `HttpContext`, который содержит информацию, специфичную для HTTP. С помощью этого контекста можно прочитать информацию заголовка HTTP
`Server`	Возвращает объект `HttpServerUtility`. Этот класс содержит ряд вспомогательных методов для кодирования и декодирования URL-адреса
`Session`	Возвращает объект `HttpSessionState` для хранения определенного состояния клиента
`User`	Возвращает объект пользователя, реализующий интерфейс `IPrincipal`. С помощью этого интерфейса можно получить имя пользователя и тип аутентификации
`SoapVersion`	Возвращает версию SOAP, используемую данной веб-службой. Версия SOAP описана перечислением `SoapProtocolVersion`

Атрибут `WebService`

Подкласс класса `WebService` должен быть помечен атрибутом `WebService`. Класс `WebServiceAttribute` имеет свойства, перечисленные в табл. 19.2.

Таблица 19.2. Свойства класса `WebServiceAttribute`

Свойство	Описание
`Description`	Описание службы, которое будет использовано в документе WSDL
`Name`	Извлекает или устанавливает имя веб-службы
`Namespace`	Извлекает или устанавливает пространство имен XML для веб-службы. Значением, используемым по умолчанию, является `http://tempuri.org`, которое вполне подходит для тестирования, но при публикации службы пространство имен потребуется изменить

Атрибут WebMethod

Все методы, доступные из веб-службы, должны быть помечены атрибутом WebMethod. Конечно, служба может содержать и другие методы, которые не помечены этим атрибутом. Вызов таких методов возможен из методов WebMethod, но не из клиентского приложения. Класс атрибута WebMethodAttribute позволяет вызывать метод из удаленных клиентов и определять, должен ли ответ подвергаться буферизации, в течение какого времени кэш должен оставаться действительным и должно ли состояние сеанса сохраняться с именованными параметрами. Свойства класса WebMethodAttribute перечислены в табл. 19.3.

Таблица 19.3. Свойства класса WebMethodAttribute

Свойство	Описание
BufferResponse	Получает или устанавливает флаг, указывающий необходимость буферизации ответа. Значение этого свойства по умолчанию равно true. При буферизации ответа только готовый пакет отправляется клиенту
CacheDuration	Устанавливает продолжительность временного интервала, в течение которого должно выполняться кэширование результата. При вторичном выполнении того же запроса только кэшированное значение будет возвращено, если запрос выполняется в течение периода, заданного этим свойством. Значение этого свойства по умолчанию равно 0, что означает отключение кэширования результата
Description	Используется при генерации справочных страниц службы для предполагаемых клиентов
EnableSession	Булевское значение, указывающее, является ли состояние сеанса допустимым. Значение этого свойства по умолчанию равно false, т.е. свойство Session класса WebService не может использоваться для хранения состояния сеанса
MessageName	По умолчанию имя сообщения совпадает с именем метода
TransactionOption	Указывает поддержку транзакций для метода. Значение этого свойства по умолчанию равно Disabled

Атрибут WebServiceBinding

Атрибут WebServiceBinding служит для пометки уровня соответствия веб-службы требованиям к взаимодействию веб-служб. Свойства этого атрибута описаны в табл. 19.4.

Таблица 19.4. Свойства класса WebServiceBinding

Свойство	Описание
ConformsTo	Устанавливается равным значению перечисления WsiProfile. WsiProfile может иметь два значения: BasicProfile1_1, когда веб-служба соответствует базовому профилю Basic Profile 1.1, или None при отсутствии какого-либо соответствия
EmitConformanceClaims	Булевское свойство, которое определяет, должны ли заявления о соответствии, указанные в свойстве ConformanceClaims, передаваться в обобщенную документацию WSDL
Name	Определяет имя привязки. По умолчанию это имя совпадает с именем веб-службы с добавлением к нему строки Soap
Location	Определяет расположение информации о привязке, например, http://www.wrox.com/DemoWebservice.asmx?wsdl
Namespace	Определяет XML-пространство имен привязки

Клиент

Для вызова метода клиент должен создать HTTP-соединение с сервером веб-службы и отправить HTTP-запрос для передачи сообщения SOAP. Вызов метода должен быть преобразован в сообщение SOAP. Все эти действия выполняются прокси-клиентом. Реализация прокси-клиента осуществляется в классе `SoapHttpClientProtocol`.

Класс `SoapHttpClientProtocol`

Класс `System.Web.Services.Protocols.SoapHttpClientProtocol` — это базовый класс прокси-клиента. Метод `Invoke()` преобразует аргументы для построения сообщения SOAP, отправляемого веб-службе. Вызываемая веб-служба определяется свойством `Url`.

Класс `SoapHttpClientProtocol` поддерживает также асинхронные вызовы с помощью методов `BeginInvoke()` и `EndInvoke()`.

Альтернативные клиентские протоколы

Вместо класса `SoapHttpClientProtocol` можно использовать и другие классы прокси-клиента. `HttpGetClientProtocol` и `HttpPostClientProtocol` просто выполняют простые HTTP-запросы `GET` или `POST`, не порождая накладных расходов, связанных с вызовом SOAP.

Классы `HttpGetClientProtocol` и `HttpPostClientProtocol` можно применять, если решение использует .NET на стороне клиента и сервера. Для поддержки других технологий должен применяться протокол SOAP.

Сравните следующий HTTP-запрос `POST` с ранее приведенным вызовом SOAP:

```
POST /WebServiceSample/Service1.asmx/ReverseString HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: длина

message=строка
```

HTTP-запрос `GET` еще короче. Недостаток запроса `GET` состоит в том, что длина отправленных параметров ограничена. Если их длина превышает 1 Кбайт, следует подумать об использовании запроса `POST`.

```
GET /WebServiceSample/Service1.asmx/ReverseString?message=строка HTTP/1.1
Host: localhost
```

Накладные расходы, связанные с классами `HttpGetClientProtocol` и `HttpPostClientProtocol`, меньше чем те, что связаны с методами SOAP. Однако недостаток этого подхода в том, что какая-либо поддержка со стороны веб-служб, действующих на других платформах, равно как и поддержка отправки чего-либо кроме простых данных отсутствует.

Создание простой веб-службы ASP.NET

В следующем практическом занятии мы создадим простую веб-службу с помощью Visual Studio.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Создание проекта веб-службы**

1. Выбрав пункт меню File⇨New⇨Project (Файл⇨Создать⇨Проект), создайте новый проект веб-службы и укажите шаблон ASP.NET Empty Web Application (Пустое веб-приложение ASP.NET), как показано на рис. 19.4. Назначьте проекту имя `WebServiceSample` и щелкните на кнопке OK.
2. Добавьте в проект новый элемент, выберите шаблон Web Service (Веб-служба) и назначьте создаваемому файлу имя `SampleService.asmx`, как показано на рис. 19.5.

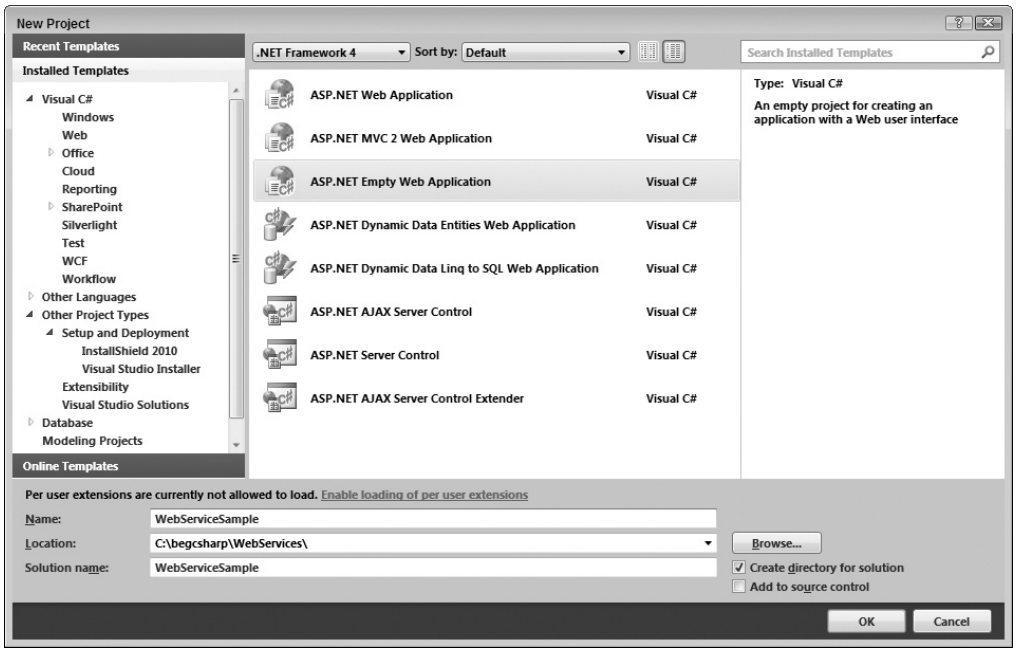


Рис. 19.4. *Создание нового проекта* ASP.NET Empty Web Application

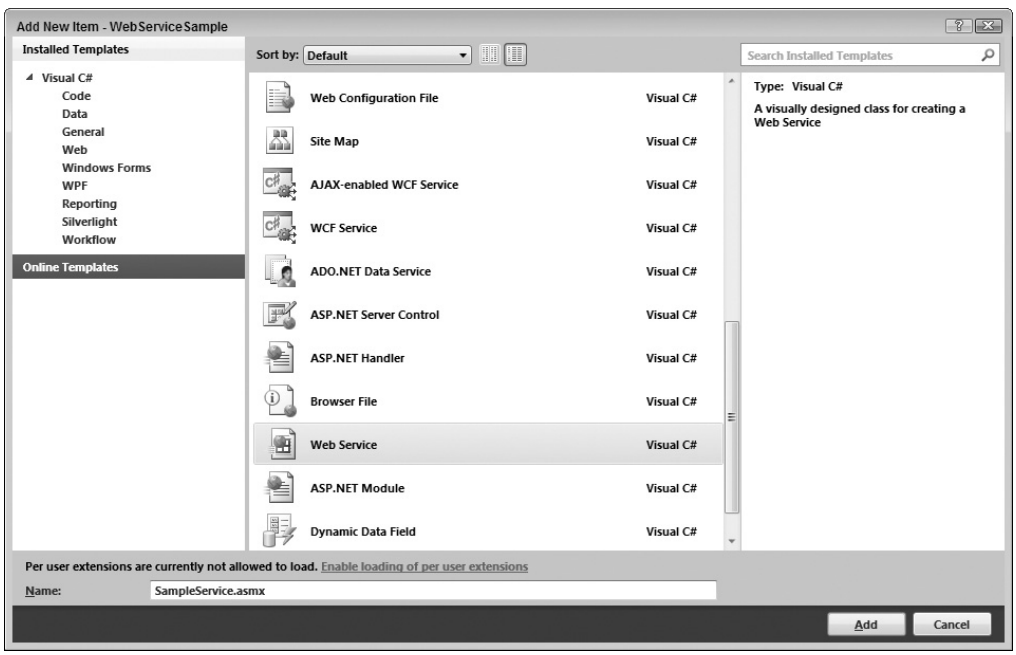


Рис. 19.5. *Добавление в проект новой веб-службы*

Описание работы

Файлы, сгенерированные шаблонами проекта и элементов, перечислены ниже.

- `SampleService.asmx`. Этот файл содержит класс веб-службы. Все веб-службы ASP.NET обозначаются расширением `.asmx`. Исходный код хранится в файле `SampleService.asmx.cs`, поскольку связанная с ним функциональность используются средой Visual Studio.
- `SampleService.asmx.cs`. Шаблон элемента генерирует в файле `SampleService.asmx.cs` класс `SampleService`, который является производным от `System.Web.Services.WebService`. Этот файл содержит также образец кода для метода веб-службы — он должен быть общедоступным и помеченным атрибутом `WebMethod`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;

namespace WebServiceSample
{
   [WebService(Namespace = "http://tempuri.org/")]
   [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
   [System.ComponentModel.ToolboxItem(false)]
   public class SampleService : System.Web.Services.WebService
   {
      [WebMethod]
      public string HelloWorld()
      {
         return "Hello World";
      }
   }
}
```

*Фрагмент кода **WebServiceSample\SampleService.asmx.cs***

Добавление веб-метода

Следующее, что потребуется сделать — это добавить метод в создаваемую веб-службу. В приведенном ниже практическом занятии мы добавим простой метод `ReverseString()`, который принимает строку и возвращает клиенту строку в обратном порядке.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Добавление метода

1. Удалите метод `HelloWorld()` и всю его реализацию. Добавьте следующий код в файл `SampleService.asmx.cs`.

```
[WebMethod]
public string ReverseString(string message)
{
   return new string(message.Reverse().ToArray());
}
```

*Фрагмент кода **WebServiceSample\SampleService.asmx.cs***

2. Модифицируйте пример кода в файле `SampleService.asmx.cs`, как показано ниже:

```
[WebService(Namespace = "http://www.wrox.com/BeginningCSharp/2010")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
```

```
// Чтобы разрешить вызов этой веб-службы из сценария с помощью ASP.NET Ajax,
// удалите метку комментария со следующей строки:
// [System.Web.Script.Services.ScriptService]
public class SampleService : System.Web.Services.WebService
```

Фрагмент кода ***WebServiceSample\SampleService.asmx.cs***

3. Скомпилируйте проект.

Описание работы

Исполняющая среда ASP.NET использует рефлексию для считывания ряда специфичных для веб-служб атрибутов, таких как `[WebMethod]`, чтобы предоставить метод в качестве операции веб-службы. Исполняющая среда ASP.NET предоставляет также WSDL-документ, описывающий службу.

Чтобы XML-элементы в сгенерированном описании веб-службы могли идентифицироваться уникальным образом, необходимо добавить пространство имен XML. В приведенном примере пространство имен `http://www.wrox.com/webservices` было добавлено в класс `Service` с указанием атрибута `[WebService]`. Естественно, можно применять любую другую строку, которая уникально идентифицирует XML-элементы, такую как URL-ссылку страницы своей организации или компании. Вовсе не обязательно, чтобы веб-ссылка существовала в действительности. Она предназначена только для уникальной идентификации. Использование пространства имен, построенного на основе веб-адреса своей организации, обеспечивает почти стопроцентную гарантию того, что ни одна другая компания не будет применять это же пространство имен.

Если пространство имен XML не будет изменено, по умолчанию будет использоваться пространство имен `http://tempuri.org`. В учебных целях это пространство имен вполне подходит, но его не следует применять для развертывания реальной веб-службы.

Тестирование веб-службы

Теперь веб-службу можно протестировать. Открытие файла `Service1.asmx` в браузере (его можно запустить из Visual Studio 2010, выбрав пункт меню Debug⇨Start Without Debugging (Отладка⇨Запустить без отладки)) ведет к отображению списка всех методов службы, как показано на рис. 19.6. В созданной нами службе имеется единственный метод — `ReverseString()`.

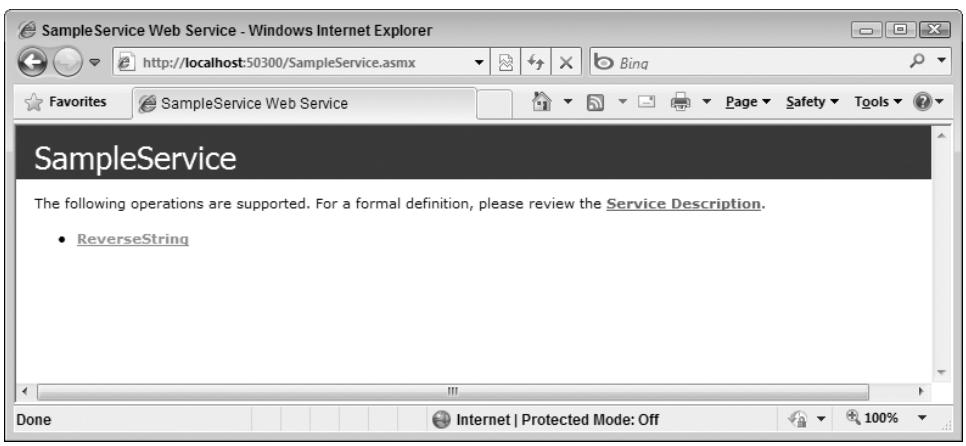


Рис. 19.6. *Список методов службы*

При выборе ссылки на метод `ReverseString` открывается диалоговое окно тестирования веб-службы. Оно содержит поля редактирования каждого параметра, который можно передать с методом. В данном случае параметр является единственным.

На этой странице можно также получить информацию о том, как будут выглядеть запросы SOAP со стороны клиента и ответы сервера (рис. 19.7). Пример демонстрирует запросы SOAP и HTTP-запросы `POST`.

Щелчок на кнопке Invoke (Вызвать) после ввода текста `Hello Web Services!` в текстовом поле приведет к получению с сервера результата, показанного на рис. 19.8.

Результат типа `string`, как и ожидалось, представляет собой введенную строку, отображенную в обратном порядке.

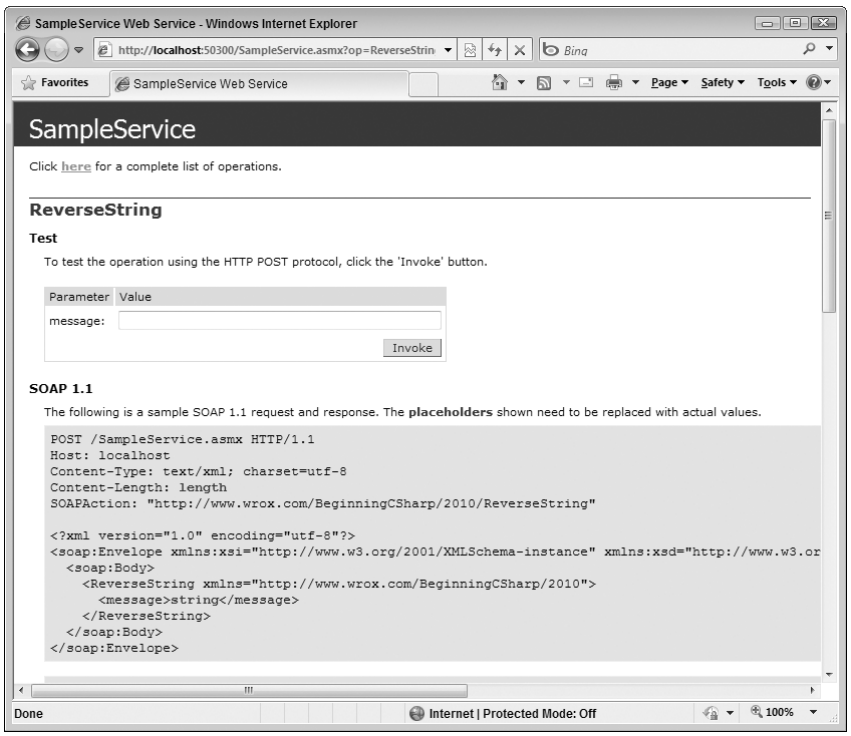


Рис. 19.7. *Пример запроса SOAP*

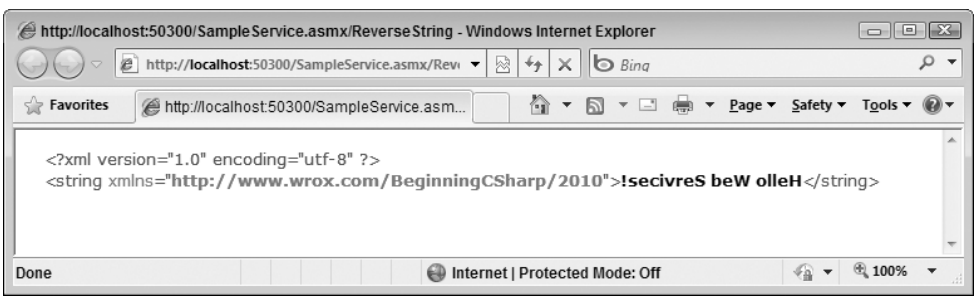


Рис. 19.8. *Тестовое обращение к веб-службе*

Реализация Windows-клиента

Тестирование прошло успешно, однако нам нужно построить Windows-клиент, который использует веб-службу. Клиент должен создавать сообщение SOAP, которое будет пересылаться по HTTP-каналу. Это сообщение не обязательно создавать самостоятельно. Среда Visual Studio 2010 создает прокси-класс, использующий HTTP-канал WCF, который "за кулисами" выполняет все необходимые действия.

Технология WCF более подробно рассматривается в главе 26.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание клиентского Windows-приложения

1. Создайте новое C#-приложение Windows Forms в существующем решении `WebServiceSample` и присвойте ему имя `WindowsFormsClient`. Добавьте на форму два текстовых поля и кнопку (рис. 19.9). Для вызова веб-службы будет использоваться обработчик события `Click` этой кнопки.

Рис. 19.9. *Пользовательский интерфейс клиентского Windows-приложения*

2. С помощью команды меню Project⇨Add Service Reference (Проект⇨Добавить ссылку на службу) добавьте ссылку на службу. В открывшемся диалоговом окне Add Service Reference (Добавление ссылки на службу) щелкните на стрелке кнопки Discover (Исследовать) и выберите опцию Services in Solution (Службы в решении). Ранее созданная служба отображается в левой панели. В отображенном слева древовидном представлении выберите `SampleServiceSoap`. Прежде чем щелкнуть на кнопке OK, измените имя в поле Namespace (Пространство имен) на `WebServicesSample`, как показано на рис. 19.10.

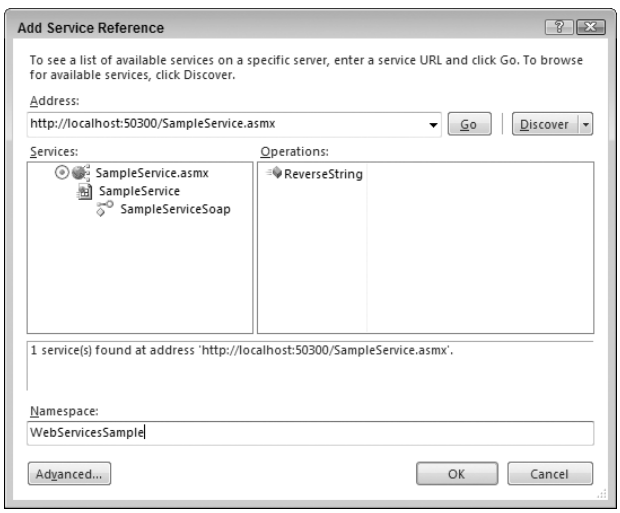


Рис. 19.10*. Добавление ссылки на службу*

3. До сих пор мы не написали ни единой строчки кода для клиента. Мы разработали небольшой интерфейс пользователя и воспользовались меню Add Service Reference, чтобы создать класс прокси-клиента. Теперь остается только связать эти два объекта. Добавьте к кнопке обработчик `button1_Click()` события `Click` и поместите в него следующие два оператора:

```
private void button1_Click(object sender, EventArgs e)
{
   var client = new WebServicesSample.SampleServiceSoapClient();
   textBox2.Text = client.ReverseString(textBox1.Text);
}
```

Фрагмент кода ***WindowsFormsClient\Form1.cs***

Описание работы

Теперь новая ссылка на службу `WebServiceSample` должна отображаться в окне Solution Explorer (рис. 19.11). Щелкните на кнопке Show All Files (Показать все файлы), чтобы увидеть документ WSDL и файл `Reference.cs`, содержащий исходный код прокси-клиента. Кнопка Show All Files — вторая слева в панели инструментов Solution Explorer. При помещении указателя мыши над кнопками всплывающие подсказки отображают информацию о назначении кнопок.

Информацию, отображаемую в Solution Explorer только после щелчка на кнопке Show All Files, удобнее просматривать в представлении классов Class View (новый класс, реализующий прокси-клиента). Этот класс преобразует вызовы методов в формат SOAP. Представление Class View (рис. 19.12) будет содержать новое пространство имен с названием, которое было определено при добавлении ссылки на службу. В данном случае было создано пространство имен `WebServiceSample`. Класс `SampleServiceSoapClient` является производным от `ClientBase<SampleServiceSoap>` и реализует метод веб-службы — `ReverseString()`.

Дважды щелкните на классе `SampleServiceSoapClient`, чтобы открыть автоматически сгенерированный файл `Reference.cs`. Рассмотрим этот сгенерированный код.

Класс `SampleServiceSoapClient` является производным от класса `ClientBase<SampleServiceSoap>`. Этот базовый класс создает сообщение SOAP в методе `Invoke()`.

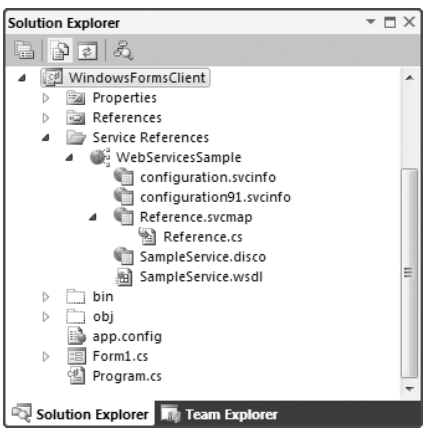


Рис. 19.11*. Новая ссылка на службу* `WebServiceSample`

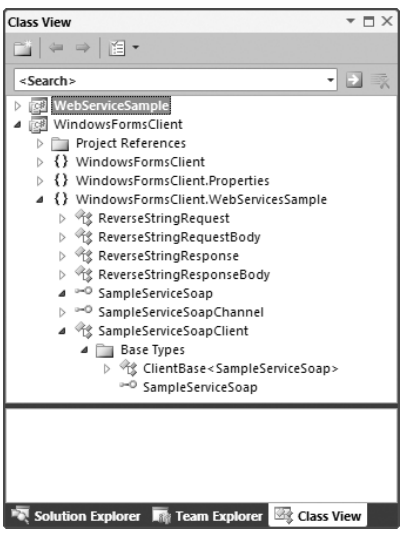


Рис. 19.12*. Новое пространство имен в представлении* Class View

SampleServiceSoap — это интерфейс, который определяет все операции, выполняемые веб-службой.

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "4.0.0.0")]
public partial class SampleServiceSoapClient : ClientBase<SampleServiceSoap>,
    SampleServiceSoap {
```

Фрагмент кода ***WindowsFormsClient\Service References\WebServicesSample\Reference.svcmap\Reference.cs***

Наиболее важным здесь является метод, предоставляемый веб-службой: ReverseString(). Этот метод имеет тот же параметр, который реализован на сервере. Реализация клиентской версии метода ReverseString() вызывает метод Invoke() базового класса SoapHttpClientProtocol. Метод Invoke() создает сообщение SOAP, используя имя метода ReverseString и параметр message. Этот метод определен в файле reference.cs.

```
public string ReverseString(string message) {
    ReverseStringRequest inValue = new ReverseStringRequest();
    inValue.Body = new ReverseStringRequestBody();
    inValue.Body.message = message;
    ReverseStringResponse retVal =
         ((SampleServiceSoap)(this)).ReverseString(inValue);
    return retVal.Body.ReverseStringResult;
}
```

Фрагмент кода ***WindowsFormsClient\Service References/WebServicesSample\Reference.svcmap\Reference.cs***

Отправка запроса SOAP по HTTP-соединению определена в автоматически созданном файле конфигурации приложения, который определяет basicHttpBinding:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="SampleServiceSoap" closeTimeout="00:01:00"
          openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
          allowCookies="false" bypassProxyOnLocal="false"
          hostNameComparisonMode="StrongWildcard" maxBufferSize="65536"
          maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
          useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384" maxBytesPerRead="4096"
            maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None" proxyCredentialType="None"
              realm="" />
            <message clientCredentialType="UserName" algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:50300/SampleService.asmx"
        binding="basicHttpBinding" bindingConfiguration="SampleServiceSoap"
      contract="WebServicesSample.SampleServiceSoap" name="SampleServiceSoap" />
    </client>
  </system.serviceModel>
</configuration>
```

Фрагмент кода ***WindowsFormsClient\app.config***

Обращение к службе производится в следующем операторе с помощью сгенерированного прокси-класса. Этот оператор создает новый экземпляр прокси-класса. Как показано в реализации конструктора, свойство `Url` устанавливается в соответствии с веб-службой:

```
var client = new WebServicesSample.SampleServiceSoapClient();
```

Фрагмент кода ***WindowsFormsClient\Form1.cs***

В результате вызова метода `ReverseString()` прокси-класса сообщение SOAP отправляется серверу, что приводит к вызову веб-службы:

```
textBox2.Text = client.ReverseString(textBox1.Text);
```

Фрагмент кода ***WindowsFormsClient\Form1.cs***

В результате выполнения программы получается вывод, подобный показанному на рис. 19.13.

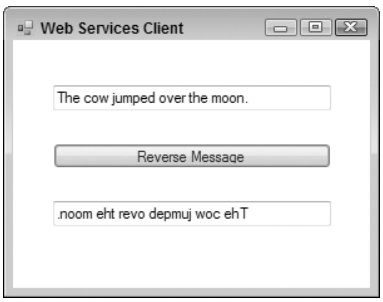


Рис. 19.13*. Результат обращения к веб-службе в клиентском Windows-приложении*

Асинхронный вызов службы

При отправке сообщения по сети всегда нужно учитывать сетевую задержку. При синхронном вызове веб-службы клиентское приложение блокируется до момента возвращения из вызова. В локальной сети это может происходить достаточно быстро, однако следует принимать во внимание инфраструктуру производственной сети.

Сообщения веб-службе можно отправлять асинхронно. Прокси-клиент создает не только синхронные методы, но и асинхронные, однако при использовании Windows-приложений ситуация носит особый характер. Поскольку каждый элемент управления Windows связан с единственным потоком, методы и свойства элементов управления Windows могут вызываться только из создающего потока. Прокси-класс .NET 4 предоставляет ряд специальных средств, как видно в сгенерированном коде прокси-объекта.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Асинхронный вызов службы

Чтобы использовать асинхронную реализацию прокси-класса, выполните следующие действия.

1. Внесите изменение в код сгенерированного прокси-класса, выбрав ссылку на службу `WebServicesSample`. Откройте контекстное меню и выберите пункт Configure Service Reference (Конфигурировать ссылку на службу). Откроется диалоговое окно Service Reference Settings (Настройки ссылки на службу), показанное на рис. 19.14.

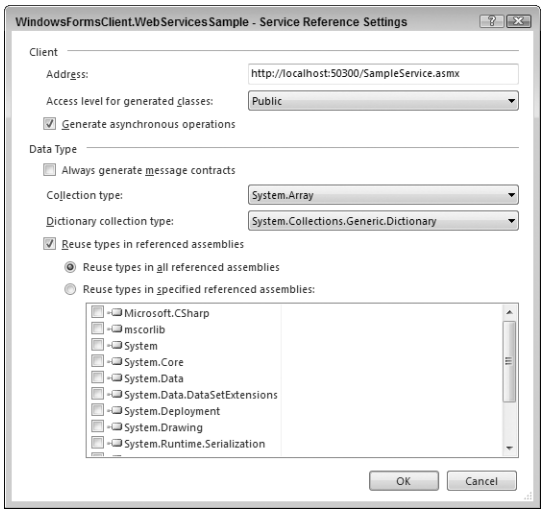


***Рис. 19.14**. Диалоговое окно* Service Reference Settings

2. В диалоговом окне Service Reference Settings отметьте флажок Generate asynchronous operations (Генерировать асинхронные операции).

3. Чтобы вызывать веб-службу асинхронно, измените реализацию метода `button1_Click` (код приведен ниже). После того как экземпляр прокси-объекта создан, добавьте к событию `ReverseStringCompleted` обработчик по имени `client_ReverseStringCompleted`. Затем вызовите асинхронный метод прокси-объекта `ReverseStringAsync` и передайте ему свойство `Text` из `textBox1`. Метод `asnyc` создает поток, который выполняет обращение к веб-службе.

```
private void button1_Click(object sender, EventArgs e)
{
   var client = new WebServicesSample.SampleServiceSoapClient();
   client.ReverseStringCompleted += client_ReverseStringCompleted;
   client.ReverseStringAsync(textBox1.Text);
}
```

Фрагмент кода ***WindowsFormsClient\Form1.cs***

4. Теперь реализуйте метод обработчика `client_ReverseStringCompleted`:

```
void client_ReverseStringCompleted(object sender,
                                   ReverseStringCompletedEventArgs e)
{
   if (e.Error != null)
   {
      textBox2.Text = e.Result;
   }
   else
   {
      MessageBox.Show(e.Error.Message);
   }
}
```

Фрагмент кода ***WindowsFormsClient\Form1.cs***

Этот метод будет вызываться по завершении вызова веб-службы. При данной реализации свойство `Result` параметра `ReverseStringComletedEventArgs` передается свойству `Text` объекта `textBox2`.

5. Теперь можно запустить клиентское приложение еще раз, чтобы протестировать асинхронный вызов службы. В реализацию веб-службы можно добавить также интервал приостановки, чтобы удостовериться, что интерфейс пользователя клиентского приложения не блокируется на время вызова веб-службы.

Описание работы

В показанном ниже фрагменте кода представлена асинхронная версия метода `ReverseString()`. Асинхронная реализация прокси-класса всегда содержит метод, который можно вызывать асинхронно, и событие, позволяющее определять, какой метод должен вызываться по завершении метода веб-службы.

Метод `ReverseStringAsync()` содержит только те параметры, которые отправляются серверу. Считывание данных, полученных от клиента, можно выполнять, назначая обработчик события событию `ReverseStringCompleted`, имеющему тип `EventHandler<ReverseStringCompletedEventArgs>`. Он представляет собой делегат, в то время как второй параметр (`ReverseStringCompletedEventArgs`) создается из выходных параметров метода `ReverseString()`. Свойство `Result` класса `ReverseStringCompletedEventArgs` содержит возвращаемые данные веб-службы. Эта реализация работает благодаря делегату `SendOrPostCallback`, который направляет вызов в нужный поток элемента управления Windows Forms.

```
public event System.EventHandler<ReverseStringCompletedEventArgs>
    ReverseStringCompleted;

public void ReverseStringAsync(string message) {
   this.ReverseStringAsync(message, null);
}

public void ReverseStringAsync(string message, object userState) {
   if ((this.onBeginReverseStringDelegate == null)) {
      this.onBeginReverseStringDelegate =
         new BeginOperationDelegate(this.OnBeginReverseString);
   }
   if ((this.onEndReverseStringDelegate == null)) {
      this.onEndReverseStringDelegate =
         new EndOperationDelegate(this.OnEndReverseString);
   }
   if ((this.onReverseStringCompletedDelegate == null)) {
      this.onReverseStringCompletedDelegate =
         new SendOrPostCallback(this.OnReverseStringCompleted);
   }
   base.InvokeAsync(this.onBeginReverseStringDelegate, new object[] {
            message}, this.onEndReverseStringDelegate,
            this.onReverseStringCompletedDelegate, userState);
   }

   private void OnReverseStringCompleted(object state) {
      if ((this.ReverseStringCompleted != null)) {
         InvokeAsyncCompletedEventArgs e =
            ((InvokeAsyncCompletedEventArgs)(state));
         this.ReverseStringCompleted(this,
           new ReverseStringCompletedEventArgs(e.Results, e.Error,
              e.Cancelled, e.UserState));
      }
   }
}

public partial class ReverseStringCompletedEventArgs : AsyncCompletedEventArgs {

   private object[] results;
```

```
    public ReverseStringCompletedEventArgs(object[] results,
       Exception exception, bool cancelled, object userState) :
          base(exception, cancelled, userState) {
       this.results = results;
    }

    public string Result {
       get {
          base.RaiseExceptionIfNecessary();
          return ((string)(this.results[0]));
       }
    }
}
```

Фрагмент кода ***WindowsFormsClient\Service References\WebServicesSample\Reference.svcmap\Reference.cs***

Реализация клиента ASP.NET

Теперь эту же службу можно использовать из клиентского приложения ASP.NET. Ссылка на службу может выполняться так же, как в приложении Windows Forms.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание клиентского приложения ASP.NET

1. Откройте ранее созданную веб-службу `WebServicesSample`.
2. Добавьте в решение новое пустое веб-приложение C# ASP.NET и назовите его `ASPNETClient`.
3. Создайте новую веб-форму `Default.aspx` и добавьте в нее два текстовых поля и кнопку, как показано на рис. 19.15.
4. Добавьте ссылку на службу `http://localhost:50300/SampleService.asmx`, как это было выполнено при построении Windows-приложения. В зависимости от конкретной конфигурации может потребоваться указать другой номер порта.
5. После того как ссылка службы добавлена, снова генерируется класс прокси-клиента. Добавьте к кнопке обработчик события `Click` и поместите в него следующий код:

```
protected void Button1_Click(object sender, EventArgs e)
{
   var client = new WebServicesSample.SampleServiceSoapClient();
   TextBox2.Text = client.ReverseString(TextBox1.Text);
}
```

Фрагмент кода ***ASPNETClient\Default.aspx.cs***

6. Скомпонуйте проект. Выберите пункт меню Debug⇨Start (Отладка⇨Запуск), чтобы запустить браузер, и введите тестовое сообщение в первом текстовом поле. Щелчок на кнопке приведет к вызову веб-службы и во втором текстовом поле отобразится сообщение в обратном порядке, как показано на рис. 19.16. При использовании решения с множеством проектов в качестве начального нужно установить проект, который требуется запустить.

Описание работы

Работа прокси-класса в этом случае ничем не отличается от созданного ранее клиентского приложения. Добавление ссылки на службу приводит к генерации прокси-класса на основе документа WSDL. Прокси-класс выполняет SOAP-запрос к службе.

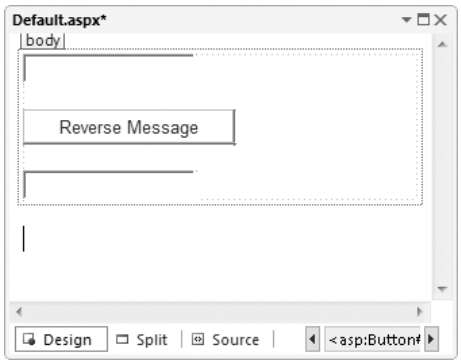


Рис. 19.15. Пользовательский интерфейс клиентского приложения ASP.NET

Рис. 19.16. Результат обращения к веб-службе в клиентском приложении ASP.NET

Передача данных

При использовании разработанной ранее простой веб-службы ей передается только простая строка. Теперь нам предстоит добавить метод для передачи информации о погоде, запрошенной из веб-службы. Это требует передачи веб-службе и из нее более сложных данных.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Обмен данными с веб-службой

1. Откройте в Visual Studio ранее созданный проект веб-службы `WebServicesSample`. Для этой веб-службы определите типы, показанные в следующем коде. Классы `GetWeatherRequest` и `GetWeatherResponse` (см. фрагменты кода ниже) определяют документы, которые должны отправляться веб-службе и из нее. Внутри этих классов применяются перечисления `TemperatureType` и `TemperatureCondition`.

   Веб-службы ASP.NET используют сериализацию XML для преобразования объектов в XML-представление. Для изменения внешнего вида сгенерированного XML-формата можно применять атрибуты из пространства имен `System.Xml.Serialization`.

```
public enum TemperatureType
{
   Fahrenheit,
   Celsius
}

public class GetWeatherRequest
{
   public string City { get; set; };
   public TemperatureType TemperatureType { get; set; };
}
```

*Фрагмент кода **WebServiceSample\GetWeatherRequest.cs***

```
public enum TemperatureCondition
{
   Rainy,
   Sunny,
   Cloudy,
   Thunderstorm
}
```

```
public class GetWeatherResponse
{
  public TemperatureCondition Condition { get; set; };
  public int Temperature { get; set; };
}
```

Фрагмент кода ***WebServiceSample\GetWeatherResponse.cs***

2. Добавьте метод `GetWeather()` веб-службы:

```
[WebMethod]
public GetWeatherResponse GetWeather(GetWeatherRequest req)
{
  var resp = new GetWeatherResponse();
  var r = new Random();
  int celsius = r.Next(-20, 50);
  if (req.TemperatureType == TemperatureType.Celsius)
    resp.Temperature = celsius;
  else
    resp.Temperature = (212 - 32) / 100 * celsius + 32;
  if (req.City == "Redmond")
    resp.Condition = TemperatureCondition.Rainy;
  else
    resp.Condition = (TemperatureCondition)r.Next(0, 3);

  return resp;
}
```

Фрагмент кода ***WebServiceSample\SampleService.asmx.cs***

Этот метод принимает данные, определенные аргументом `GetWeatherRequest`, и возвращает данные, определенные аргументом `GetWeatherResponse`. Внутри реализации осуществляется возврат произвольных погодных условий. Для генерации произвольных погодных условий используется класс `Random` из пространства имен `System`.

3. После построения веб-службы создайте новый проект на основе шаблона Windows Forms Application и назовите приложение `WeatherClient`.
4. Измените главное окно, как показано на рис. 19.17. Оно содержит два переключателя, которые позволяют выбирать температурную шкалу (по Цельсию или по Фаренгейту), и текстовое поле для ввода названия города. Щелчок на кнопке Get Weather (Получить информацию о погоде) приводит к вызову веб-службы, а результат отображается в текстовых полях Weather Condition (Погодные условия) и Temperature (Температура).

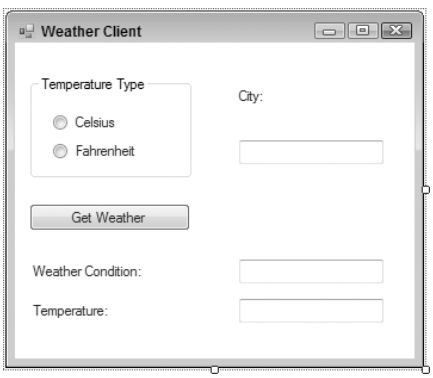


Рис. 19.17*. Пользовательский интерфейс для тестирования обмена данными с веб-службой*

Типы, имена и значения свойства Text элементов управления перечислены в следующей таблице:

Элемент управления	Имя	Значение свойства Text
GroupBox	groupBox1	Temperature Type
RadioButton	radioButtonCelsius	Celsius
RadioButton	radioButtonFahrenheit	Fahrenheit
Label	labelCity	City
TextBox	textCity	
Button	buttonGetWeather	Get Weather
Label	labelWeatherCondition	Weather Condition
Label	labelTemperature	Temperature
TextBox	textWeatherCondition	
TextBox	textTemperature	

5. Добавьте ссылку на веб-службу, подобно тому, как это было сделано в предшествующих проектах клиентских приложений. Назначьте ссылке имя WeatherService.
6. Импортируйте пространство имен WeatherClient.WeatherService, содержащее клиентское приложение.
7. Используя диалоговое окно Properties (Свойства) кнопки, добавьте к кнопке buttonGetWeather обработчик события Click по имени OnGetWeather().
8. Добавьте следующую реализацию метода OnGetWeather():

```
private void OnGetWeather(object sender, EventArgs e)
{
   var req = new GetWeatherRequest();
   if (radioButtonCelsius.Checked)
      req.TemperatureType = TemperatureType.Celsius;
   else
      req.TemperatureType = TemperatureType.Fahrenheit;
   req.City = textCity.Text;

   var client = new SampleServiceSoapClient();
   GetWeatherResponse resp = client.GetWeather(req);
   textWeatherCondition.Text = resp.Condition.ToString();
   textTemperature.Text = resp.Temperature.ToString();
}
```

*Фрагмент кода **WeatherClient\Form1.cs***

Эта реализация создает объект GetWeatherRequest, который определяет запрос, отправленный веб-службе. Обращение к веб-службе осуществляется вызовом метода GetWeather(). Этот метод возвращает объект GetWeatherResponse, содержащий значения, которые считываются для отображения в интерфейсе пользователя.

9. Запустите клиентское приложение. Введите название города и щелкните на кнопке Get Weather. Если повезет, в окне отобразится информация о реальной погоде (рис. 19.18).

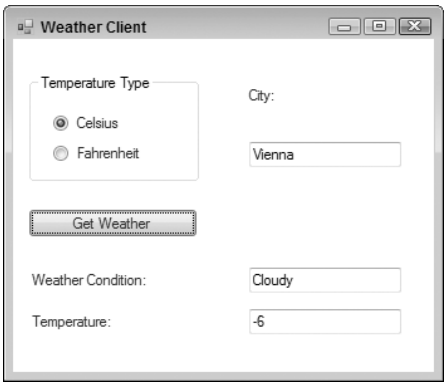


***Рис. 19.18**. Клиентское приложение взаимодействует с веб-службой, сообщающей информацию о погоде*

Описание работы

При передаче данных из среды ASP.NET и в нее веб-служба использует сериализацию XML. Применение сериализации XML ведет к сериализации всех общедоступных свойств и полей. Классы `GetWeatherRequest` и `GetWeatherResponse` используют общедоступные свойства. Чтобы использовать сериализацию XML, класс должен быть общедоступным, и должен существовать общедоступный конструктор по умолчанию. Если никакой конструктор в класс не добавляется (как в случае классов `GetWeatherRequest` и `GetWeatherResponse`), компилятор создает используемый по умолчанию общедоступный конструктор, который инициализирует все поля-члены класса. Типы значений инициализируются значением `0`, а типы ссылок — значением `null`.

Классы атрибутов, определенные в пространстве имен `System.Xml.Serialization`, могут использоваться для индивидуальной настройки XML-результата сериализации. Класс атрибута `XmlIgnoreAttribute` служит для игнорирования членов класса. Класс атрибута `XmlElementAttribute` можно применять для переименования сериализованного XML-элемента, а класс `XmlAttributeAttribute` — для сериализации XML-атрибута, а не элемента.

При использовании с операциями службы типа, доступного для XML-сериализации, в документ WSDL добавляется информация схемы XML, которая применяется при создании типов для клиентского приложения во время добавления ссылки на службу.

Резюме

В этой главе вы узнали, что собой представляют веб-службы, и кратко ознакомились с применяемыми вместе с ними протоколами. Для отыскания и запуска веб-служб можно использовать следующие возможности.

- **Описание.** Документ WSDL описывает методы и аргументы.
- **Вызов.** Независимые от платформы вызовы методов выполняются с использованием протокола SOAP.

В этой главе было показано, насколько просто создать веб-службы с помощью Visual Studio, где класс `WebService` применяется для определения определенных методов посредством атрибута `WebMethod`. Создание клиента, который использует веб-службы, столь же просто, как и создание веб-служб — достаточно в проект клиентского приложения добавить ссылку на веб-службу и воспользоваться прокси-объектом. Основная часть клиентского приложения — класс `SoapHttpClientProtocol`, который преобразует вызов метода в сообщение SOAP. Созданный прокси-клиент предоставляет как асинхронные, так и син-

хронные методы. При использовании асинхронных методов интерфейс клиента не блокируется до завершения работы метода веб-службы. Было также показано, как создавать специальные классы, которые позволяют передавать более сложные данные. Следующая глава посвящена развертыванию веб-приложений.

Упражнения

Следующие упражнения помогут применить приобретенные в этой главе знания для создания новой веб-службы, отвечающей за бронирование мест в кинотеатре.

1. Создайте новую веб-службу `CinemaReservation`.
2. Для определения данных, отправляемых веб-службе и из нее, требуются классы `ReserveSeatRequest` и `ReserveSeatResponse`. Класс `ReserveSeatRequest` нуждается в члене `Name` (Имя) типа `string` для отправки имени и в двух членах типа `int` для отправки запроса на бронирования места, определенного с помощью `Row` (Ряд) и `Seat` (Место). Класс `ReserveSeatResponse` определяет данные, отправляемые клиенту — т.е. имя, на которое выполняется бронирование, и номер в действительности забронированного места.
3. Создайте веб-метод `ReserveSeat`, который требует наличия параметра `ReserveSeatRequest` и возвращает параметр `ReserveSeatResponse`. В реализации веб-службы можно использовать объект `Cache` (см. главу 18) для запоминания уже забронированных мест. Если запрошенное место свободно, приложение должно возвратить это место и зарезервировать его в объекте `Cache`. Если место занято, приложение должно выбрать соседнее свободное место. Для хранения мест в объекте `Cache` применяйте двумерный массив, как было показано в главе 5.
4. Создайте клиентское Windows-приложение, которое использует веб-службу для бронирования мест в кинотеатре.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Создание веб-служб с помощью ASP.NET	Веб-службы ASP.NET могут быть созданы в веб-проекте ASP.NET. Служба определяется с помощью атрибутов `WebService` и `WebMethod`.
Вызов веб-служб	При использовании клиентского приложения для вызова операций, выполняемых веб-службой, прокси-объект может быть создан с помощью выбора пункта меню Add⇨Service Reference (Добавить⇨Ссылка на службу) в Solution Explorer. Добавление ссылки службы ведет к применению документа WSDL и созданию прокси-класса.
Асинхронный вызов веб-служб	Используя расширенные возможности ссылки на службу, можно создавать асинхронные методы для асинхронного вызова веб-службы. Метод с префиксом `Async` принимает входные параметры метода веб-службы. По завершении вызова службы генерируется событие, посредством которого осуществляется прием выходных параметров.
Передача данных между веб-службами	Для передачи веб-службе данных сложных типов можно создать специальный класс. XML-сериализация используется для преобразования объектов в сообщения, передаваемые по сети.

20

Развертывание веб-приложений

В ЭТОЙ ГЛАВЕ...

- Конфигурирование IIS для веб-приложений ASP.NET
- Копирование веб-сайтов Visual Studio
- Публикация веб-приложений
- Создания пакетов программы установки Windows для веб-приложений

В предыдущих двух главах рассказывалось о том, как разрабатывать веб-приложения и веб-службы с помощью ASP.NET. Для всех этих типов приложений существуют различные варианты развертывания, такие как копирование веб-страниц, публикация веб-сайта и создание программы установки. В этой главе речь пойдет о преимуществах и недостатках каждого из этих вариантов, а также о том, как они применяются на практике.

Компонент IIS

Компонент IIS (Internet Information Services — информационные службы Интернета) не нужно устанавливать для разработки веб-приложений в Visual Studio 2010, потому что данная среда располагает собственным веб-сервером, который называется Visual Web Development Server. Это простой веб-сервер, функционирующий только на локальном компьютере. Поэтому в производственной системе для запуска веб-приложений необходимо использовать все-таки IIS.

В среде Windows 7 Home Edition компонент IIS не доступен. В других версиях Windows 7 его можно устанавливать точно так же, как и другие компоненты Windows. В частности, для этого требуется просто открыть окно панели управления, щелкнуть на ссылке Programs (Программы), отыскать категорию Programs and Features (Программы и компоненты) со ссылкой Turn Windows features on or off (Включение или отключение компонентов Windows), щелкнуть на этой ссылке и в списке компонентов Windows выбрать компонент Internet Information Services (Службы IIS). При желании можно попросить установить IIS в системе системного администратора.

Исполняющая среда ASP.NET должна обязательно быть сконфигурирована с IIS для того, чтобы в ней можно было запускать веб-приложения ASP.NET. Чтобы удостовериться в надлежащей конфигурации исполняющей среды ASP.NET, проверьте сопоставления обработчиков в инструменте IIS Manager (рис. 20.1). Если компонент IIS установлен, этот инструмент будет доступен в категории Administrative Tools (Администрирование) в виде элемента Internet Information Services (IIS) Manager (Диспетчер служб IIS). Если категория Administrative Tools не была сконфигурирована на доступ непосредственно из меню Start (Пуск), выберите пункт меню Control Panel⇨Administrative Tools (Панель управления⇨Администрирование).

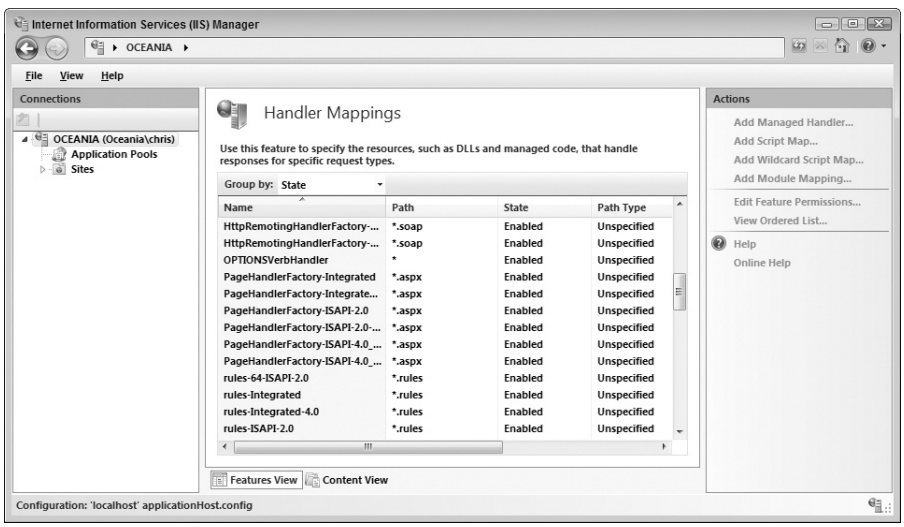


Рис. 20.1. *Сопоставления обработчиков в IIS Manager*

В окне Internet Information Services (IIS) Manager дважды щелкните на элементе Handler Mappings (Сопоставления обработчиков). Просмотрев информацию, которая появится после этого, можно увидеть, что путь `*.aspx` сконфигурирован несколько раз. В окне отобразится несколько конфигураций .NET Framework, а также собственная и управляемая конфигурации. Конфигурация `IsapiModule` расширения `.aspx` определяет собственную конфигурацию — класс .NET по имени `System.Web.UI.PageHandlerFactory`, который выполняет обработку запросов.

Если сопоставления обработчиков для исполняющей среды ASP.NET не сконфигурированы в системе, можно запустить команду `aspnet_regiis -i`, чтобы установить необходимые расширения файлов и модули с помощью IIS.

Главным процессом IIS является `inetinfo.exe`. Он выполняется с высокими привилегиями от имени учетной записи `System`. При сконфигурированном модуле `IsapiModule` запрос ASPX-файла будет перенаправляться рабочему процессу (`w3wp.exe`). Для запуска различных версий исполняющей среды .NET могут быть сконфигурированы разные рабочие процессы. Можно также настраивать учетные данные пользователя, от имени которого выполняется данный процесс, и указывать параметры удаления ненужных файлов.

Конфигурирование IIS

Компонент IIS должен быть обязательно соответствующим образом сконфигурирован перед тем, как можно будет запускать с его помощью какие-то веб-приложения. В следующем практическом занятии демонстрируется пример создания веб-сайта с помощью инструмента Internet Information Services (IIS) Manager. Первым делом этому веб-сайту необходим виртуальный каталог, т.е. каталог, который будет использовать клиент при получении доступа к веб-приложению. Например, в `http://server/mydirectory` виртуальным каталогом будет `mydirectory`. Виртуальный каталог совершенно не зависит от физического каталога, отвечающего за хранение файлов на диске. Например, для `mydirectory` физический каталог вполне может выглядеть как `D:\someotherdirectory`.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание нового пула приложений

1. Запустите инструмент IIS Manager (рис. 20.2). Найти ее можно в окне панели управления, в разделе Administrative Tools (Администрирование).

Рис. 20.2. *Окно диспетчера IIS*

2. В древовидном представлении выберите узел Application Pools (Пулы приложений), щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Add Application Pool (Добавить пул приложений).
3. Откроется диалоговое окно Add Application Pool (Добавление пула приложений), показанное на рис. 20.3. Введите `Beginning Visual C# App Pool` в текстовом поле Name (Имя) и выберите .NET Framework v4.0.30109 (Платформа .NET Framework, версия v4.0.30109) в поле .NET Framework version (Версии среды .NET Framework). Щелкните на кнопке OK. В этом окне можно сконфигурировать версию исполняющей среды .NET. Для ASP.NET версий 3.5, 3.0 или 2.0 можно использовать одинаковый номер версии .NET Framework — `2.0.50727`.

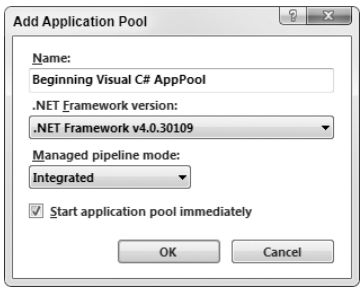


Рис. 20.3. *Диалоговое окно Add Application Pool*

4. После создания пула приложений можно настроить дополнительные параметры в диалоговом окне Advanced Settings (Дополнительные параметры), показанном на рис. 20.4. В нем следует указать идентификационные данные, с которыми должен выполняться процесс, а также то, должны ли в системе с многоядерным процессором или с несколькими процессорами использоваться только конкретные процессоры, и какое количество рабочих процессов должно выполняться в данном пуле. Диалоговое окно Advanced Settings доступно после выбора созданного пула приложений либо в категории Actions (Действия) в правой части окна Internet Information Services (IIS) Manager, либо через контекстное меню.

Описание работы

Пулы приложений позволяют использовать для разных веб-сайтов различные версии исполняющей среды ASP.NET, а также создавать для них разные учетные записи пользователей и предусматривать различную степень стабильности.

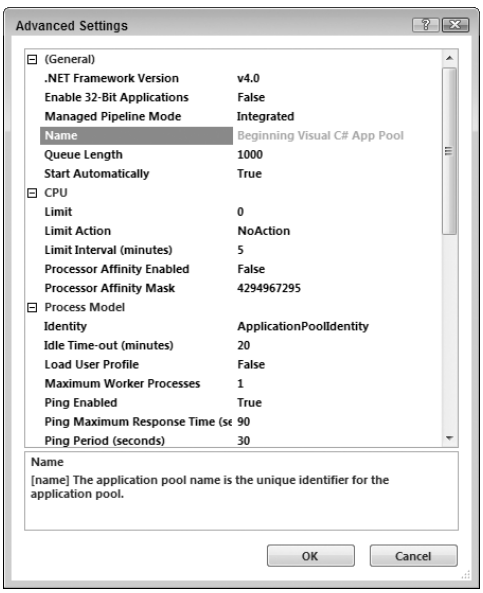


Рис. 20.4. *Диалоговое окно Advanced Settings*

После настройки пула приложений можно переходить к созданию нового веб-приложения, как показано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание нового веб-приложения

1. В окне IIS Manager выберите в древовидном представлении узел Default Web Site (Веб-сайт по умолчанию).
2. Щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Add Application (Добавить приложение). После этого на экране появится диалоговое окно Add Application (Добавление приложения), показанное на рис. 20.5.
3. Введите физический путь к веб-сайту и псевдоним `BeginningVCSharpWebsite`. Выберите только что созданный пул `Beginning Visual C# App Pool`.

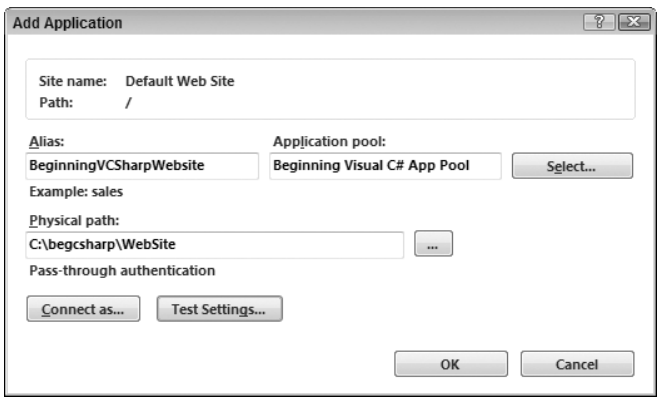


Рис. 20.5. *Диалоговое окно* Add Application

4. Щелкните на кнопке OK.

Теперь веб-приложение сконфигурировано, а это значит, что веб-сайт можно начинать использовать для копирования или публикации веб-приложений из Visual Studio.

Копирование веб-сайта

В Visual Studio 2010 можно копировать файлы из исходного веб-сайта в удаленный. Исходным веб-сайтом называется веб-сайт того веб-приложения, которое было открыто в Visual Studio. Доступ к нему осуществляется либо из локальной файловой системы, либо из IIS — в зависимости о того, как веб-приложение создавалось. Доступ к удаленному веб-сайту, на который требуется скопировать файлы, может осуществляться как с помощью файловой системы, так и по протоколу FTP или посредством FrontPage Server Extensions в IIS.

Копирование файлов допускается в обоих направлениях, т.е. как из исходного веб-сайта в удаленный, так и наоборот. В следующем практическом занятии демонстрируется пример использования Visual Studio для копирования только что созданного веб-приложения в веб-сайт, который был сконфигурирован ранее в этой главе.

Меню копирования веб-сайтов Visual Studio доступно только из веб-сайта, но не из окна Web Project *(Веб-проект).*

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Копирование веб-сайта

1. При работе под управлением Windows 7 или Windows Vista запустите Visual Studio 2010 с повышенными административными правами, потому что для копирования веб-сайта на локальный сервер IIS необходимо иметь права администратора. В случае использования Windows XP можете запустить Visual Studio обычным образом при условии, если вход в систему был выполнен от имени учетной записи с правами администратора.
2. С помощью пункта меню File⇨New Web Site (Файл⇨Создать веб-сайт) создайте новый веб-сайт и выберите шаблон ASP.NET Web Site (Веб-сайт ASP.NET). В качестве места размещения этого веб-сайта укажите локальную файловую систему. В результате будет создан пример сайта, который использует несколько страниц и стилей.
3. Выберите пункт меню Website⇨Copy Web Site (Веб-сайт⇨Копировать веб-сайт). Откроется диалоговое окно, показанное на рис. 20.6.

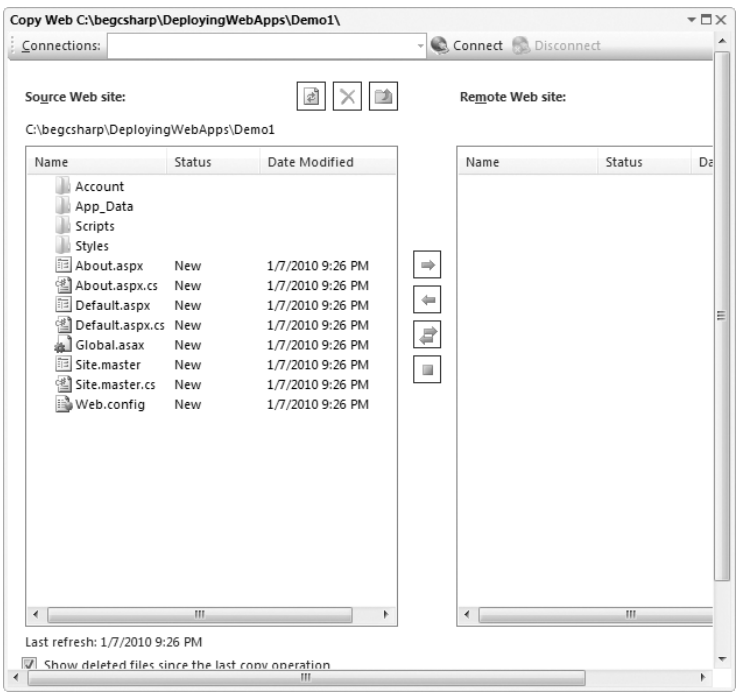


Рис. 20.6. *Окно копирования веб-сайта*

4. Щелкните на кнопке Connect (Подключиться), расположенной в верхней части этого окна. Откроется диалоговое окно Open Web Site (Открытие веб-сайта).
5. В этом окне можно выбирать файлы для копирования в следующие места: локальная файловая система, локальный сервер IIS, сайты FTP и удаленные сайты (с установленным компонентом FrontPage Server Extensions). Щелкните здесь на кнопке Local IIS (Локальный сервер IIS) и выберите ранее созданный веб-сайт `BegVCSharpWebsite` (рис. 20.7). В версии Windows Home Edition файлы можно копировать только в локальную файловую систему, потому что сервер IIS в этой версии не доступен.

Рис. 20.7. *Диалоговое окно* Open Web Site

6. В списке Source Web site (Исходный веб-сайт) выберите файлы, которые хотите скопировать из исходного веб-сайта в удаленный веб-сайт.
7. Щелкните на кнопке Copy Selected Files (Скопировать выбранные файлы). Эта кнопка расположена между представлениями Source Web Site (Исходный веб-сайт) и Remote Web Site (Удаленный веб-сайт) и содержит изображение стрелки. При наведении курсора мыши на любую из расположенных здесь кнопок отображается подсказка с описанием назначения кнопки. Направление стрелки на них указывает на то, в каком направлении будут копироваться файлы: из исходного в удаленный сайт или наоборот. Кнопка со стрелками, указывающими в обоих направлениях, подразумевает выполнение проверки на предмет того, какие файлы являются более новыми, и копирование в другой сайт только их.
8. После этого все выбранные файлы будут скопированы в новый веб-сайт. Для обращения к скопированному веб-сайту можно открыть браузер и ввести адрес `http://localhost/BeginningVCSharpWebsite`.

Описание работы

С помощью инструмента копирования сайта (Copy Web Site) можно также выбирать и файлы для копирования из удаленного веб-сайта в исходный. Кнопка Synchronize Selected Files (Синхронизировать выбранные файлы) с изображением стрелок, указывающих в обоих направлениях, позволяет копировать из удаленного веб-сайта в исходный и из исходного в удаленный только более новые файлы. Эта опция очень полезна при наличии общего веб-сервера, на котором другие разработчики синхронизируют файлы. Выполнение синхронизации в обоих направлениях позволяет легко осуществлять копирование своих более новых файлов на общий веб-сервер, а файлов с удаленного веб-сервера коллег — в свой локальный сайт.

Если выполняется только копирование файлов, нельзя быть уверенным в том, что они будут успешно компилироваться. Компиляция выполняется при обращении браузера к файлам. Однако существует утилита командной строки `aspnet_compiler.exe`, с помощью которой можно выполнять предварительную компиляцию веб-сайта.

Достаточно ввести команду `aspnet_compiler -v /BeginningVCSharpWebsite` и веб-сайт `BeginningVCSharpWebsite` будет предварительно скомпилирован. Благодаря этому, первому же пользователю не придется дожидаться окончания компиляции ASPX-страниц, поскольку они будут уже скомпилированы.

Найти эту утилиту можно в каталоге исполняющей среды .NET.

Публикация веб-приложения

Инструмент Web Project среды Visual Studio 2010 предоставляет также возможность публикации веб-приложения. Этот вариант наиболее подходит в тех случаях, когда вы не являетесь владельцем самостоятельной службы IIS, и веб-приложение нужно опубликовать в сайте поставщика.

Существует несколько различных возможностей публикации с помощью Visual Studio 2010.

- Публикация в файловой системе.
- Публикация на сервере с установленным компонентом FrontPage Server Extensions.
- Использование протокола FTP.
- Использование нового средства Visual Studio 2010 под названием публикация 1-Click. Опция 1-Click доступна только для партнеров хостинга, которые поддерживают эту функциональность, но список таких партнеров уже достаточно велик и его легко найти.

В следующем практическом занятии показано, как использовать новое средство публикации Visual Studio для публикации веб-приложения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Публикация веб-приложения

1. Откройте веб-проект `EventRegistrationWeb`, созданный в главе 18.
2. Откройте вкладку настроек проекта Package/Publish (Пакет/Публикация), как показано на рис. 20.8. Проверьте местоположение, в котором будет создан пакет публикации. Щелкните на ссылке Open Settings (Открыть настройки), расположенной рядом с флажком Include all Databases configured in Deploy SQL Tab (Включить все базы данных, сконфигурированные на вкладке SQL-код развертывания).

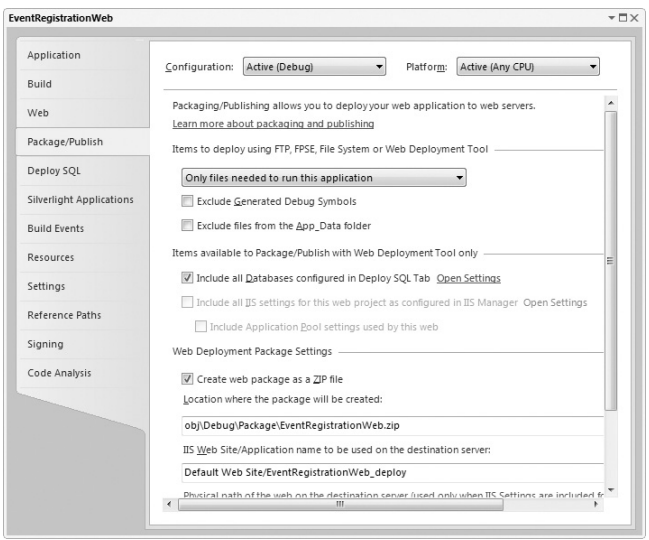


Рис. 20.8. *Вкладка* Package/Publish *окна свойств проекта*

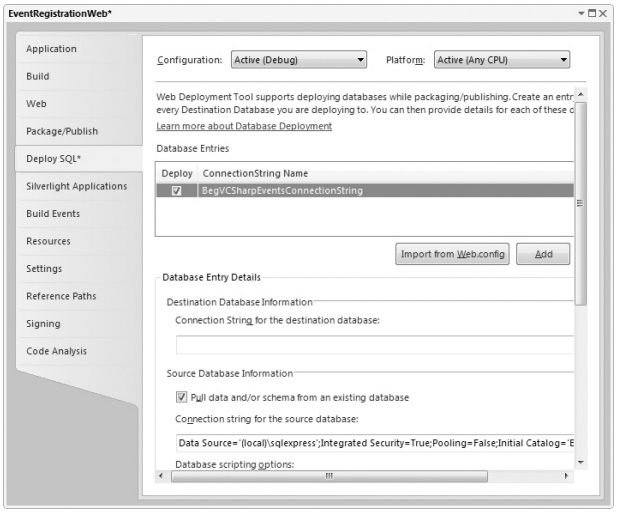


Рис. 20.9. *Вкладка* Deploy SQL *окна свойств проекта*

3. Настройки вкладки Deploy SQL (SQL-код развертывания) показаны на рис. 20.9. Щелкните на кнопке Import from Web.config (Импортировать из `Web.config`), чтобы импортировать строку подключения к базе данных. Можно развернуть также базу данных, на которую ссылается строка подключения. Проверьте правильность остальных настроек. В этом окне можно также определить строку подключения к серверу целевой базы данных, на котором должны быть записаны данные и схема базы данных.
4. В меню Build (Компоновать) Visual Studio выберите пункт Publish (Опубликовать). Откроется диалоговое окно Publish Web (Опубликовать веб-приложение), показанное на рис. 20.10. Проверьте настройки метода публикации MSDeploy Publish (Публикация MSDeploy). Опция 1-Click Publish (Публикация 1-Click) доступна при использовании нескольких поставщиков хостинга. Щелкните на ссылке Click Here (Щелкните здесь), чтобы найти компанию, предоставляющую услуги хостинга в вашем регионе. Вместо использования этой опции метод публикации можно изменить на File System (Файловая система). Разумеется, если поставщик хостинга поддерживает эту опцию, ее можно использовать для публикации веб-приложения.
5. При выборе метода публикации в файловой системе откроется диалоговое окно, показанное на рис. 20.11. Введите имя локального каталога в поле Target Location (Целевое расположение) и щелкните на кнопке Publish (Опубликовать).
6. Откройте целевое расположение в проводнике Windows и проверьте опубликованные файлы.

Программа установки Windows

Для установки веб-приложения можно также создать программу установки Windows. Создание установочных программ требуется только в том случае, если в приложении необходимо использовать разделяемые сборки. Преимущество подхода с созданием установочных программ состоит в том что, в случае его применения виртуальный каталог конфигурируется с помощью IIS и, следовательно, создавать его вручную не требуется. Человеку, устанавливающему веб-приложение, достаточно просто запустить программу `setup.exe`, и весь процесс установки будет выполнен автоматически. Разумеется, для ее запуска обязательно нужны привилегии администратора.

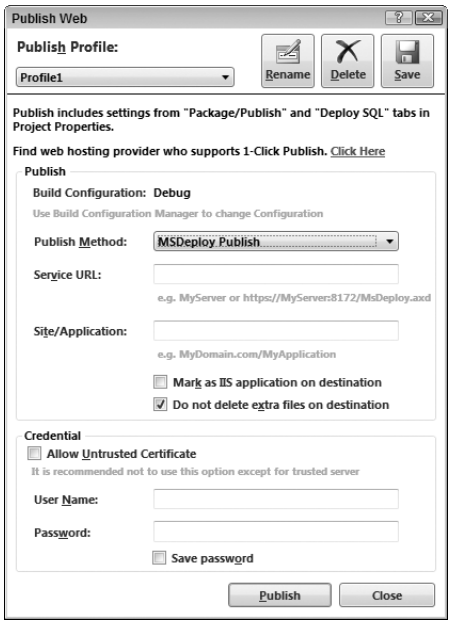


Рис. 20.10. *Диалоговое окно* Publish Web

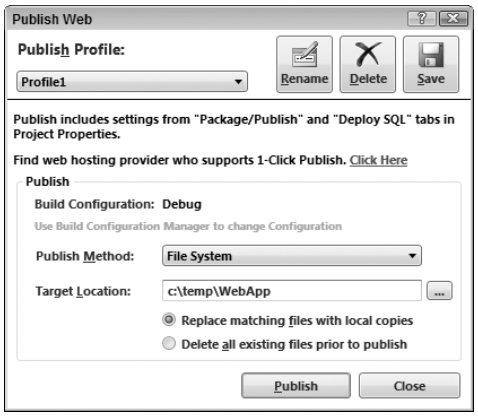


Рис. 20.11. *Выбор метода публикации* File System

Создание установочной программы

В Visual Studio 2010 для создания установочных программ веб-приложений предлагается проект специального типа, который называется Web Setup Project (Проект программы установки веб-приложения). В его составе доступны такие редакторы, как File System Editor (Редактор файловой системы), Registry Editor (Редактор реестра), File Types Editor (Редактор типов файлов), User Interface Editor (Редактор пользовательского интерфейса), а также Custom Action Editor (Редактор специальных действий) и Launch Conditions Editor (Редактор условий запуска). Об этих редакторах уже рассказывалось в главе 17 при рассмотрении Windows-приложений, поэтому здесь будут упомянуты только те из них, которые должны использоваться для веб-приложений.

В следующем практическом занятии демонстрируется пример создания установочной программы для веб-приложения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание установочной программы

1. Откройте в Visual Studio 2010 веб-приложение `EventRegistrationWeb`, созданное в главе 18.
2. Добавьте в ранее созданное решение новый проект типа Web Setup Project, как показано на рис. 20.12. Назначьте ему имя `EventRegistrationWebSetup` и щелкните на кнопке OK.
3. Если редактор File System Editor (Редактор файловой системы) не было открыт после создания проекта, откройте его. Выберите опцию File System on Target Machine (Файловая система на целевом компьютере), а затем выберите пункт меню Project⇨Add⇨Project Output (Проект⇨Добавить⇨Вывод проекта). Затем в диалоговом окне Project Output (Вывод проекта) выберите опцию Content Files of Web Application (Файлы содержимого веб-приложения) и щелкните на кнопке OK.

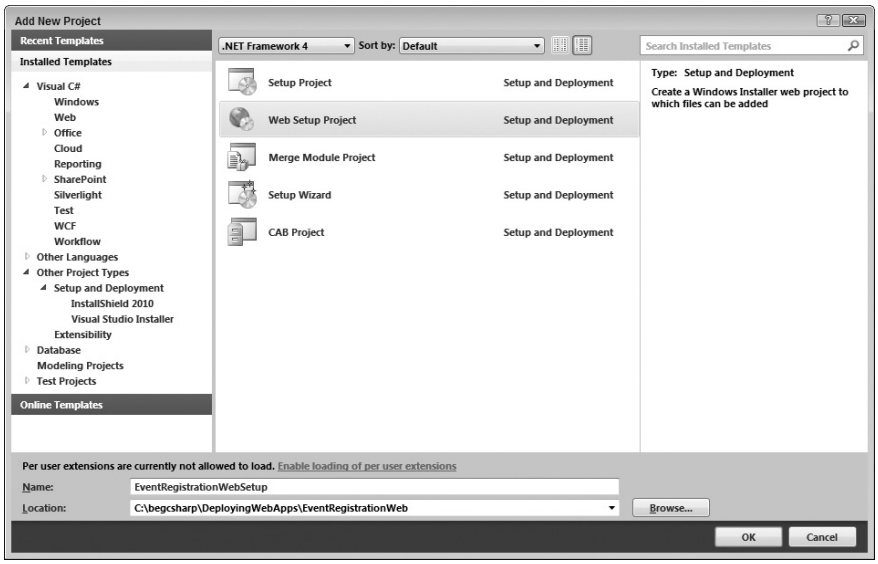


***Рис. 20.12**. Добавление нового проекта программы установки `EventRegistrationWebSetup`*

4. Вернувшись в окно File System Editor, в поле Web Application Folder (Папка веб-приложения) выберите папку веб-приложения. После этого можно сконфигурировать веб-приложение с помощью редактора свойств. Свойства веб-приложения описаны в табл. 20.1.

Таблица 20.1. Свойства веб-приложения

Свойство	Описание
`AllowDirectoryBrowsing`	Представляет собой опцию конфигурации IIS. В случае установки его в `true` разрешает отыскивать файлы на веб-сайте путем просмотра. По умолчанию для него устанавливается значение `false`
`AllowReadAccess`	По умолчанию для этого свойства устанавливается значение `true`. Для обращения к ASPX-страницам обязательно требуется доступ по чтению
`AllowScriptSourceAccess`	За счет установки этого свойства в `false` по умолчанию получать доступ к исходным файлам сценария запрещено
`AllowWriteAccess`	Получать доступ для записи по умолчанию тоже запрещено
`DefaultDocument`	Указывает домашнюю страницу веб-сайта, например, `Default.aspx`
`ExecutePermissions`	По умолчанию для этого свойства устанавливается значение `vsdepScriptsOnly`, которое разрешает получать доступ к страницам ASP.NET, но не разрешает запускать на сервере специальные исполняемые файлы. Чтобы на сервере можно было запускать специальные исполняемые файлы, это свойство необходимо установить в `vsdepScriptsAndExecutables`

Окончание табл. 20.1

Свойство	Описание
`LogVisits`	Если установлено в `true`, обеспечивается регистрация попыток доступа клиентов
`VirtualDirectory`	Указывает имя виртуального каталога, который конфигурируется с помощью IIS

5. Откройте редактор Launch Condition Editor, выбрав пункт меню View⇨Editor⇨Launch Conditions (Вид⇨Редактор⇨Условия запуска). Условия запуска позволяют указывать, какие продукты должны быть обязательно установлены на целевой системе перед выполнением установки веб-приложения.
6. Проверьте сконфигурированные условия запуска. Конфигурация Search for IIS (Поиск сервера IIS) предусматривает выполнение проверки на предмет наличия в целевой системе установленной копии IIS, для чего просматривается ключ реестра `\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters` и выясняется ее версия. Следующее условие запуска IIS, добавляемое по умолчанию, гарантирует, что в системе установлена служба IIS версии не ниже 5.1.

```
(IISMAJORVERSION>= "#5" ANDIISMINORVERSION>= "#1") ORIISMAJORVERSION>= "#6"
```

7. Скомпилируйте программу установки, выбрав пункт меню Build⇨Build Event RegistrationWebSetup (Компоновка⇨Компоновка `EventRegistrationWebSetup`).
8. После этого в каталоге установочного проекта можно будет найти файл `setup.exe` и установочный пакет по имени `EventRegistrationWebSetup.msi`.

Установка веб-приложения

Веб-приложение можно установить, запуская программу `setup.exe`, как описано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Установка веб-приложения**

1. Дважды щелкните на файле `setup.exe` для запуска процесса установки веб-приложения. Когда откроется диалоговое окно User Account Control (Контроль учетных записей пользователей), щелкните на кнопке Yes (Да), чтобы разрешить изменения. Откроется окно со страницей приветствия мастера установки (рис. 23.13). Щелкните в нем на кнопке Next (Далее).
2. На странице Select Installation Address (Выбор адреса установки), которая показана на рис. 20.14, измените имя виртуального каталога в поле Virtual Directory (Виртуальный каталог) так, чтобы оно отличалось от того, которое уже было сконфигурировано с помощью IIS. Выберите ранее созданный пул приложений (`Beginning Visual C# App Pool`) и щелкните на кнопке Next.
3. Подтвердите необходимость установки, щелкнув на кнопке Next в диалоговом окне Confirm Installation (Подтверждение установки). Откроется диалоговое окно с индикатором хода работ по установке.
4. После успешной установки откроется страница Installation Complete (Установка завершена), как показано на рис. 20.15. Щелкните на кнопке Close (Закрыть).
5. Теперь можно попробовать запустить веб-сайт из нового виртуального каталога.

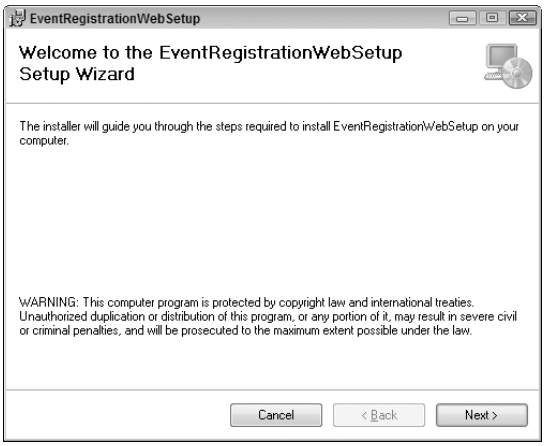


***Рис. 20.13**. Страница приветствия мастера установки*

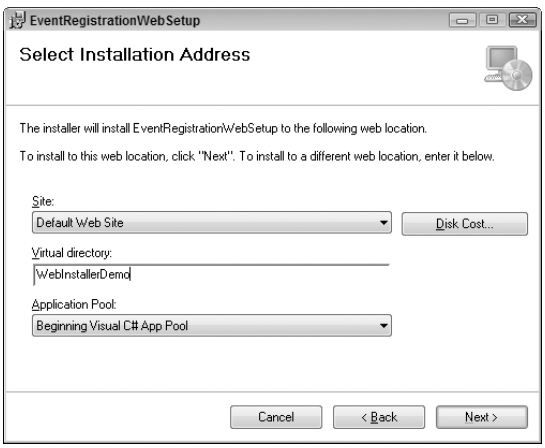


***Рис. 20.14**. Страница Select Installation Address мастера установки*

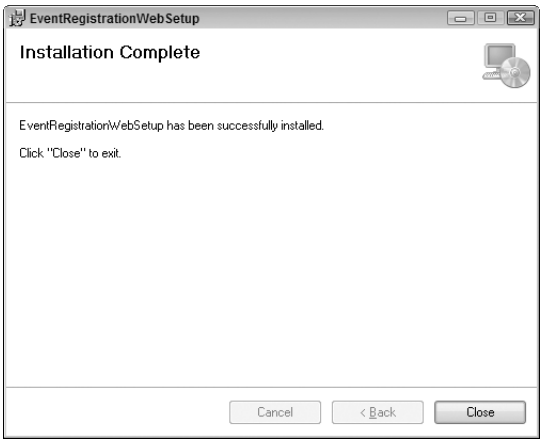


***Рис. 20.15**. Страница Installation Complete мастера установки*

Резюме

В этой главе были описаны различные варианты для развертывания веб-приложений. Инструмент Copy Web Site позволяет копировать файлы на веб-серверы с использованием общих файловых ресурсов, протокола FTP и серверного компонента FrontPage Server Extensions. Было показано, что синхронизация файлов может осуществляться в обоих направлениях. Публикация веб-приложений — новая функциональная возможность Visual Studio 2010, которая позволяет использовать публикацию с помощью единственного щелчка (1-Click), публикацию в каталоге, сервер FTP или IIS с установленным компонентом FrontPage Server Extensions. При наличии в IIS административных прав можно запустить программу установки, которая создает новое приложение внутри IIS. Установочные проекты не только копируют страницы и сборки ASP.NET, но и создают виртуальный каталог внутри IIS.

Упражнения

1. В чем состоит разница между копированием и публикацией веб-приложения? В каких случаях следует делать первое, а в каких — второе?
2. Когда предпочтительнее создавать установочную программу, чем выполнять копирование сайта?
3. Опишите различные возможности публикации веб-проекта и предъявляемые к ним требования.
4. Опубликуйте веб-службу из главы 19 в виртуальный каталог, сконфигурированный с помощью IIS.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Конфигурирование IIS	Чтобы веб-приложение ASP.NET можно было запускать в IIS, этот компонент должен быть сконфигурирован. Сопоставление обработчиков определяет классы, которые вызываются при запросе файлов со специфическими расширениями (вроде `.aspx`). Конфигурация пула приложений определяет используемую версию исполняющей среды .NET.
Копирование веб-сайта	Простой вариант публикации веб-приложения — использование копирования. В среде Visual Studio меню копирования веб-сайтов доступно только для веб-сайтов, но не для веб-проектов. Копирование файлов может осуществляться с компьютера разработчика на сервер и в обратном направлении.
Публикация веб-приложения	При использовании хостинга веб-сайтов публикация веб-приложений может быть упрощена с помощью нового средства Visual Studio 2010 — публикации 1-Click. Меню публикации позволяет также публиковать веб-приложения на серверах FTP и в файловой системе. Можно публиковать также базу данных, используемую приложением.
Программа установки Windows для веб-приложений	Если веб-приложение требуется создать внутри сервера IIS, это можно сделать с помощью установочной программы. Шаблон Web Setup Project (Проект программы установки веб-приложения) создает файл программы установки Windows, который не только копирует содержимое веб-приложения, но и создает виртуальный каталог.

ЧАСТЬ IV

Доступ к данным

В ЭТОЙ ЧАСТИ...

21

Данные файловой системы

В ЭТОЙ ГЛАВЕ...

- Что такое поток и как .NET использует потоковые классы для доступа к файлам
- Как использовать объект `File` для манипуляции файловой структурой
- Как читать и записывать в файл
- Как читать и записывать форматированные данные в файлы
- Как читать и записывать сжатые файлы
- Как выполнять сериализацию и десериализацию объектов
- Как наблюдать за изменениями в файлах и каталогах

Чтение и запись файлов — важнейшие аспекты многих приложений .NET. В этой главе будет показано, как использовать основные классы для создания, чтения и записи файлов, вместе с поддерживающими классами для манипуляций файловой системой в коде C#. Хотя все эти классы здесь детально не рассматриваются, будет дано достаточное представление о концепциях и основах этих операций.

Файлы могут быть замечательным средством хранения данных между экземплярами одного и того же приложения; их также можно применять для передачи данных между разными приложениями. Конфигурационные установки пользователя и приложения могут сохраняться и извлекаться при следующем запуске приложения. Текстовые файлы с разделителями, такие как файлы со строками, разделенными запятыми, применяются во многих унаследованных системах, и для того, чтобы взаимодействовать с этими системами, нужно знать, как работать с такими данными. Как будет показано, .NET Framework предоставляет все необходимые инструменты для эффективного использования файлов в приложениях.

Потоки

Весь ввод и вывод в .NET Framework подразумевает использование *потоков*. Поток (stream) — это абстрактное представление *последовательного устройства*. Последовательное устройство (serial device) — это нечто такое, что хранит данные в линейной манере и точно таким же образом обеспечивает доступ к ним: по одному байту за раз. Это устройство может быть дисковым файлом, сетевым каналом, местом в памяти или любым другим объектом, поддерживающим чтение и запись в линейном режиме. Сохранение устройства абстрактным означает, что лежащие в основе источник/приемник данных могут быть скрыты. Такой уровень абстракции обеспечивает многократное использование кода и позволяет писать более обобщенные процедуры, потому что нет необходимости заботиться о действительной специфике передачи данных. Таким образом, сходный код может быть передан и повторно использован, когда приложение читает из входного файлового потока, сетевого входного потока или любого другого вида потока. Поскольку физические механизмы каждого устройства можно игнорировать, имея дело с файловым потоком, не приходится беспокоиться, например, о головках жесткого диска или выделении памяти.

Существуют два типа потоков.

- **Выходные**. Выходные потоки используются, когда данные пишутся в некоторое внешнее место назначения, которым может быть физический дисковый файл, местоположение в сети, принтер или другая программа. Понимание программирования для потоков открывает множество замечательных возможностей. Материал этой главы сконцентрирован на данных файловой системы, поэтому мы сосредоточимся только на записи в дисковые файлы.
- **Входные**. Входные потоки используются для чтения данных в память или переменные, к которым может обращаться программа. Наиболее часто используемой формой входного потока, с которой вы работали до сих пор, была клавиатура. Входной поток может поступать почти из любого источника, но данная глава сосредоточена на чтении дисковых файлов. Концепции, применимые к чтению/записи дисковых файлов, подходят для большинства устройств, так что вы получите базовое представление о потоках и увидите в действии испытанный подход, используемый во многих ситуациях.

Классы ввода и вывода

Пространство имен `System.IO` содержит почти все классы, с которыми вы ознакомитесь в этой главе. `System.IO` содержит классы для чтения данных из файлов и записи их в файлы, и вы можете ссылаться на эти пространства имен в приложении C#, чтобы

получить доступ к этим классам без полной квалификации имен типов. Как показано на рис. 21.1, в `System.IO` содержится относительно немного классов, но вы будете работать только с первичными классами, необходимыми для файлового ввода и вывода.

Классы, описанные в этой главе, перечислены в табл. 21.1.

Таблица 21.1. Классы ввода-вывода, рассматриваемые в настоящей главе

Класс	Описание
`File`	Статический служебный класс, предоставляющий множество статических методов для перемещения, копирования и удаления файлов
`Directory`	Статический служебный класс, предоставляющий множество статических методов для перемещения, копирования и удаления каталогов
`Path`	Служебный класс, используемый для манипулирования путевыми именами
`FileInfo`	Представляет физический файл на диске, имеет методы для манипулирования этим файлом. Для любого чтения или записи в этот файл должен быть создан объект `Stream`
`DirectoryInfo`	Представляет физический каталог на диске и имеет методы для манипулирования этим каталогом
`FileSystemInfo`	Служит базовым классом для `FileInfo` и `DirectoryInfo`, обеспечивая возможность работы с файлами и каталогами одновременно, используя полиморфизм
`FileStream`	Представляет файл, который может быть записан, прочитан или то и другое. Этот файл может быть записан или прочитан как синхронно, так и асинхронно
`StreamReader`	Читает символьные данные из потока и может быть создан с использованием класса `FileStream` в качестве базового
`StreamWriter`	Пишет символьные данные в поток и может быть создан с использованием класса `FileStream` в качестве базового
`FileSystemWatcher`	Наиболее сложный класс, который рассматривается в этой главе. Используется для мониторинга файлов и каталогов и представляет события, которые приложение может перехватить, когда в этих объектах происходят какие-то изменения. Этой функциональности всегда недоставало в программировании для Windows, но теперь .NET Framework значительно облегчает задачу реагирования на события файловой системы

Вы также ознакомитесь с пространством имен `System.IO.Compression`, которое позволяет читать и записывать в сжатые файлы, применяя либо сжатие GZIP, либо схему сжатия Deflate.

- `DeflateStream`. Представляет поток, в котором данные автоматически сжимаются при записи или автоматически распаковываются при чтении. Сжатие обеспечивается с помощью алгоритма Deflate.
- `GZipStream`. Представляет поток, в котором данные автоматически сжимаются при записи или автоматически распаковываются при чтении. Сжатие обеспечивается алгоритмом GZIP.

И, наконец, вы изучите сериализацию объектов с использованием пространства имен `System.Runtime.Serialization` и его дочерних пространств имен. Прежде всего, мы рас-

смотрим класс `BinaryFormatter` из пространства имен `System.Runtime.Serialization.Formatters.Binary`, который позволяет сериализовывать объекты в потоки в двоичном виде, а затем десериализовывать их.

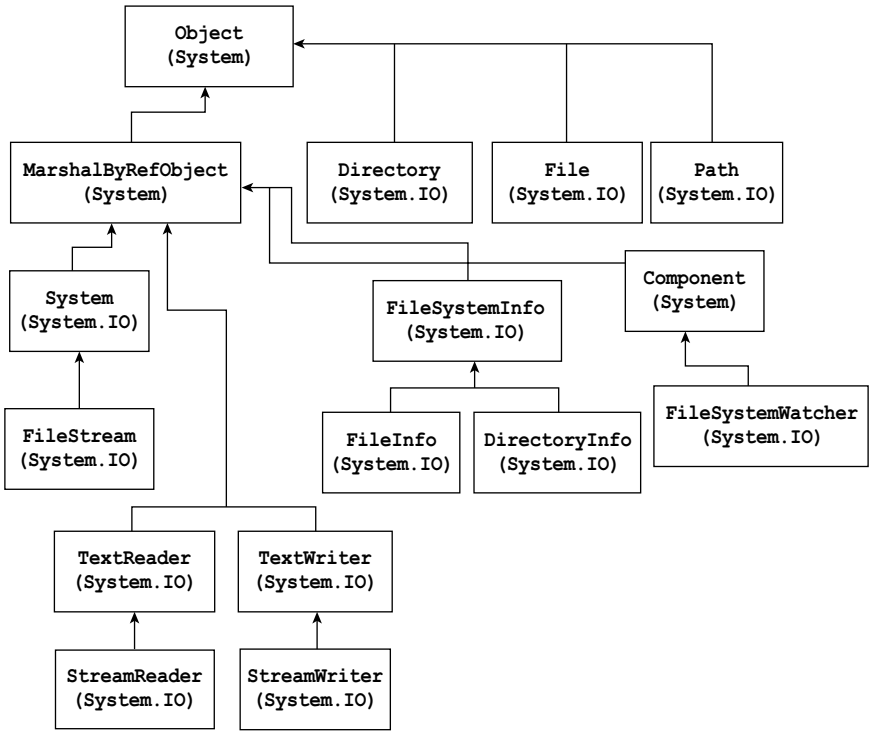


Рис. 21.1. *Классы ввода-вывода*

Классы `File` и `Directory`

Служебные классы `File` и `Directory` предлагают множество статических методов для удивительно исчерпывающего манипулирования файлами и каталогами. Эти методы обеспечивают возможность перемещения файлов, запроса и обновления атрибутов, а также создания объектов `FileStream`. Как известно из главы 8, статические методы могут вызываться на классах без создания их экземпляров.

Некоторые из наиболее полезных статических методов класса `File` перечислены в табл. 21.2.

Таблица 21.2. Некоторые статические методы класса `File`

Метод	Описание
`Copy()`	Копирует файл из исходного местоположения в целевое
`Create()`	Создает файл по указанному пути
`Delete()`	Удаляет файл
`Open()`	Возвращает объект `FileStream`, находящийся по указанному пути
`Move()`	Перемещает указанный файл в новое место. Для файла в новом местоположении можно указывать другое имя

Некоторые полезные статические методы класса `Directory` описаны в табл. 21.3.

Таблица 21.3. Некоторые статические методы класса `Directory`

Метод	Описание
`CreateDirectory()`	Создает каталог с указанным путем
`Delete()`	Удаляет указанный каталог и все файлы внутри него
`GetDirectories()`	Возвращает массив объектов `string`, представляющих имена каталогов внутри указанного каталога
`EnumerateDirectories()`	Подобен `GetDirectories()`, но возвращает коллекцию `IEnumerable<string>` имен каталогов
`GetFiles()`	Возвращает массив объектов `string`, представляющих имена файлов в указанном каталоге
`EnumerateFiles()`	Подобен `GetFiles()`, но возвращает коллекцию `IEnumerable<string>` имен файлов
`GetFileSystemEntries()`	Возвращает массив объектов `string`, представляющих имена файлов и каталогов внутри указанного каталога
`EnumerateFileSystemEntries()`	Подобен `GetFileSystemEntries()`, но возвращает коллекцию `IEnumerable<string>` имен файлов и каталогов
`Move()`	Перемещает указанный каталог в новое место. Для каталога в новом местоположении можно указывать другое имя

Три метода `EnumerateXxx()` появились в версии .NET 4. При наличии большого количества файлов и каталогов они обеспечивают лучшую производительность, чем их аналоги `GetXxx()`.

Класс `FileInfo`

В отличие от `File`, класс `FileInfo` не является статическим и не имеет статических методов. Этот класс полезен только при создании его экземпляров. Объект `FileInfo` представляет файл на диске или сетевом ресурсе, и его можно создавать, указывая путь к файлу:

```
FileInfo aFile = new FileInfo(@"C:\Log.txt");
```

Поскольку на протяжении настоящей главы вы будете работать со строковым представлением пути файла, а это означает наличие большого количества символов \ в строках, напоминаем, что символ @, предваряющий показанную выше строку, означает, что строка будет интерпретирована как литерал. Поэтому символ \ будет трактоваться как \, а не как символ начала управляющей последовательности. Без префикса @ понадобилось бы применять \\ вместо \, чтобы избежать интерпретации этого символа как начала управляющей последовательности. В этой главе мы постоянно будем использовать префикс @ со строками.

Конструктору `FileInfo` можно также передать имя каталога, хотя в практическом смысле это не слишком удобно, поскольку требует инициализации базового класса для `FileInfo`, которым является `FileSystemInfo`, информацией каталога, но ни один из методов `FileInfo` или свойств, относящихся к файлам, не будет использован.

Многие методы, представленные классом `FileInfo`, подобны методам класса `File`, но поскольку `File` — статический класс, он требует строкового параметра, указывающего местоположение файла для каждого вызова метода. Поэтому приведенные ниже вызовы делают одно и то же:

```
FileInfo aFile = new FileInfo("Data.txt");
if (aFile.Exists)
  Console.WriteLine("File Exists");      // файл существует
if (File.Exists("Data.txt"))
  Console.WriteLine("File Exists");      // файл существует
```

В этом коде производится проверка существования файла `Data.txt`. Обратите внимание, что здесь не указывается никакой информации о каталоге; это означает, что наличие файла проверяется только в текущем рабочем каталоге. Этот каталог — тот, что содержит приложение, вызвавшее данный код. Чуть позже мы рассмотрим это немного подробнее (в разделе "Путевые имена и относительные пути").

Большинство методов `FileInfo` отражают методы `File`, но на свой лад. В большинстве случаев не имеет значения, какую технику вы предпочтете, тем не менее, следующие критерии могут помочь выбрать более подходящий подход.

- Методы статического класса `File` имеет смысл использовать, если вы осуществляете единственный вызов метода; такой вызов будет быстрее, потому что .NET Framework не нужно проходить процесс создания экземпляра нового объекта с последующим вызовом его метода.
- Если ваше приложение выполняет несколько операций с файлом, то более оправдано создать экземпляр объекта `FileInfo` и пользоваться его методами; это позволит сэкономить время, потому что объект будет уже ссылаться на корректный файл в файловой системе, в то время как статический класс вынужден находить его каждый раз заново.

Класс `FileInfo` также предоставляет свойства лежащего в основе файла, часть из которых позволяет выполнять его обновление. Многие из этих свойств унаследованы от `FileSystemInfo` и потому применимы как к `File`, так и к `Directory`. Свойства класса `FileSystemInfo` перечислены в табл. 21.4.

Таблица 21.4. Свойства класса `FileSystemInfo`

Свойство	Описание
`Attributes`	Получает или устанавливает атрибуты текущего файла или каталога, используя перечисление `FileAttributes`
`CreationTime`, `CreationTimeUtc`	Получает или устанавливает дату и время текущего файла. Доступно в версиях универсального синхронизированного времени (UTC) и не UTC
`Extension`	Извлекает расширение файла. Это свойство доступно только для чтения
`Exists`	Определяет, существует ли файл. Это доступное только для чтения абстрактное свойство переопределено в `FileInfo` и `DirectoryInfo`
`FullName`	Извлекает полный путь файла. Это свойство доступно только для чтения
`LastAccessTime`, `LastAccessTimeUtc`	Получает или устанавливает дату и время последнего обращения к текущему файлу. Доступно в версиях универсального синхронизированного времени (UTC) и не UTC
`LastWriteTime`, `LastWriteTimeUtc`	Получает или устанавливает дату и время последней записи в текущий файл. Доступно в версиях универсального синхронизированного времени (UTC) и не UTC
`Name`	Извлекает полный путь файла. Это доступное только для чтения абстрактное свойство переопределено в `FileInfo` и `DirectoryInfo`

Свойства, специфичные для `FileInfo`, описаны в табл. 21.5.

Таблица 21.5. Свойства класса `FileInfo`

Свойство	Описание
`Directory`	Извлекает объект `DirectoryInfo`, представляющий каталог, который содержит текущий файл. Это свойство доступно только для чтения
`DirectoryName`	Возвращает путь каталога файла. Это свойство доступно только для чтения
`IsReadOnly`	Ссылка на атрибут файла "только для чтения". Это свойство также доступно через `Attributes`
`Length`	Получает размер файла в байтах в виде значения типа `long`. Это свойство доступно только для чтения

Объект `FileInfo` сам по себе не представляет поток. Чтобы прочитать или записать в файл, необходимо создать объект `Stream`. Объект `FileInfo` помогает в этом, предлагая несколько методов, возвращающих экземпляры объектов `Stream`.

Класс `DirectoryInfo`

Класс `DirectoryInfo` работает точно так же, как класс `FileInfo`. Это объект, представляющий отдельный каталог на машине. Подобно классу `FileInfo`, многие вызовы методов дублируются в `Directory` и `DirectoryInfo`. Все инструкции по выбору между `File` и `FileInfo` также применимы к методам `DirectoryInfo`.

- Если необходим единственный вызов, используйте класс `Directory`.
- Если необходима последовательность вызовов, используйте экземпляр объекта `DirectoryInfo`.

Класс `DirectoryInfo` наследует большинство свойств от `FileSystemInfo`, как и `FileInfo`, хотя эти свойства оперируют каталогами, а не файлами. Существуют также два специфичных для `DirectoryInfo` свойства, показанные в табл. 21.6.

Таблица 21.6. Свойства класса `DirectoryInfo`

Свойство	Описание
`Parent`	Извлекает объект `DirectoryInfo`, представляющий каталог, который содержит текущий каталог. Это свойство доступно только для чтения
`Root`	Извлекает объект `DirectoryInfo`, представляющий корневой каталог текущего тома, например, `C:\`. Это свойство доступно только для чтения

Путевые имена и относительные пути

При указании путевого имени в коде .NET можно использовать как абсолютные, так и относительные путевые имена. *Абсолютный* путь явно специфицирует файл или каталог из известного местоположения, такого как привод C:. Примером этого может служить `C:\Work\LogFile.txt` — этот путь точно определяет местоположение файла, без какой-либо неоднозначности.

Относительный путь определен относительно некоторого начального местоположения. При использовании относительных путевых имен не нужно указывать привод или некоторое известное местоположение. Это применялось раньше, когда указыва-

лось, что текущий рабочий каталог будет начальной точкой (поведение по умолчанию для относительных путевых имен). Например, если приложение запускается в каталоге C:\Development\FileDemo и использует относительный путь LogFile.txt, это указывает на файл C:\Development\FileDemo\LogFile.txt. Чтобы перейти вверх по иерархии каталогов, используется строка .. Таким образом, путь ..\Log.txt указывает на файл C:\Development\Log.txt.

Как упоминалось ранее, рабочий каталог изначально устанавливается там, откуда запущено приложение. Когда вы ведете разработку в VS или VCE, это означает, что приложение находится несколькими каталогами ниже созданной папки проекта. Обычно оно располагается в каталоге *Имя_Проекта*\bin\Debug. Чтобы обратиться к файлу в корневой папке проекта, потребуется перейти на два каталога выше, указав префикс пути ..\..\ — вы еще не раз встретитесь с этим на протяжении настоящей главы.

При необходимости рабочий каталог можно определить с использованием Directory.GetCurrentDirectory() или же установить его в новый путь с помощью Directory.SetCurrentDirectory().

Объект FileStream

Объект FileStream представляет поток, указывающий на дисковый файл или сетевой путь. В то время как класс предоставляет методы для чтения и записи байт в файл, чаще всего для выполнения этих функций вы будете использовать StreamReader или StreamWriter. Причина в том, что класс FileStream оперирует байтами и байтовыми массивами, а классы Stream — символьными данными. Работать с символьными данными проще, но некоторые операции, например, произвольный доступ к файлу (доступ к данным в некоторой точке посреди файла), могут осуществляться только посредством объекта FileStream. Далее в этой главе вы узнаете об этом подробнее.

Существует несколько способов создания объекта FileStream. Конструктор имеет множество различных перегрузок, но простейшая из них принимает два аргумента: имя файла и значение перечисления FileMode:

```
FileStream aFile = new FileStream(<имя_файла>, FileMode.<Член>);
```

Перечисление FileMode содержит ряд членов, которые указывают способ открытия или создания файла. Вскоре эти способы будут продемонстрированы. Другой часто используемый конструктор выглядит так:

```
FileStream aFile = new FileStream(<имя_файла>, FileMode.<Член>, FileAccess.<Член>);
```

Третий параметр — член перечисления FileAccess — задает способ спецификации назначения потока. Члены перечисления FileAccess описаны в табл. 21.7.

Таблица 21.7. Члены перечисления FileAccess

Член	Описание
Read	Открывает файл только для чтения
Write	Открывает файл только для записи
ReadWrite	Открывает файл только для чтения или записи

Попытка выполнить действие, отличное от указанного членом перечисления FileAccess, приведет к генерации исключения. Это свойство часто используется как способ варьирования доступа пользователя к файлу на основе его уровня авторизации.

В версии конструктора FileStream, не использующего параметр-перечисление FileAccess, применяется значение по умолчанию, которым является FileAccess.ReadWrite.

Члены перечисления `FileMode` показаны в табл. 21.8. Что в действительности происходит, когда каждое из этих значений используется, зависит от того, ссылается ли указанное имя файла на существующий файл. Обратите внимание, что в этой таблице упоминается позиция в файле, на которую установлен поток при его создании; эта тема будет подробнее раскрыта в следующем разделе. Если не установлено иначе, поток устанавливается в начало файла.

Таблица 21.8. Члены перечисления `FileMode`

Член	Поведение при существующем файле	Поведение при отсутствии файла
`Append`	Файл открыт, поток установлен в конец файла. Может использоваться только в сочетании с `FileAccess.Write`	Создается новый файл. Может использоваться только в сочетании с `FileAccess.Write`
`Create`	Файл уничтожается и на его месте создается новый	Создается новый файл
`CreateNew`	Генерируется исключение	Создается новый файл
`Open`	Файл открывается, поток позиционируется в начало файла	Генерируется исключение
`OpenCreate`	Файл открывается, поток позиционируется в начало файла	Создается новый файл
`Truncate`	Файл открывается и очищается. Поток позиционируется в начало файла. Исходная дата создания файла остается неизменной	Генерируется исключение

Оба класса — `File` и `FileInfo` — предоставляют методы `OpenRead()` и `OpenWrite()`, облегчающие создание объектов `FileStream`. Первый открывает файл с доступом только для чтения, а второй позволяет доступ только для записи. Эти методы являются сокращениями, так что вам не нужно предоставлять всю необходимую информацию в форме параметров конструктора `FileStream`. Например, следующая строка кода открывает файл `Data.txt` с доступом только для чтения:

```
FileStream aFile = File.OpenRead("Data.txt");
```

Следующий код выполняет ту же функцию:

```
FileInfo aFileInfo = new FileInfo("Data.txt");
FileStream aFile = aFileInfo.OpenRead();
```

Позиция в файле

Класс `FileStream` поддерживает внутренний указатель файла, который указывает на местоположение внутри файла, где произойдет следующая операция чтения или записи. В большинстве случаев, когда файл открывается, указатель установлен на начало файла, но данный указатель можно модифицировать. Это позволяет приложению читать или записывать в любом месте внутри файла, что, в свою очередь, дает возможность организовать произвольный доступ к файлу с перепрыгиванием непосредственно в определенное место внутри файла. В результате удается сэкономить массу времени при работе с очень большими файлами, поскольку можно мгновенно перемещаться в необходимое место.

Метод, реализующий эту функциональность — `Seek()` — принимает два параметра. Первый параметр указывает, насколько далеко в байтах нужно переместить указатель файла. Второй параметр специфицирует, откуда начинать отсчет, в форме значения из

перечисления SeekOrigin. Перечисление SeekOrigin содержит три значения: Begin, Current и End.

Например, следующая строка переместит указатель файла к восьмому байту файла, начиная с самого первого байта в этом файле:

```
aFile.Seek(8, SeekOrigin.Begin);
```

Следующая строка переместит указатель файла на два байта вперед, начиная с текущей позиции. Если ее выполнить сразу после предыдущей строки, то указатель файла теперь будет находиться на десятом байте в файле:

```
aFile.Seek(2, SeekOrigin.Current);
```

Когда вы читаете или пишете в файл, файловый указатель также изменяется. После чтения 10 байт файловый указатель будет установлен на байт, находящийся после десятого прочитанного байта.

Можно также задавать отрицательное значение смещения, которое может быть скомбинировано со значением перечисления SeekOrigin.End, чтобы работать недалеко от конца файла. Следующий оператор переносит указатель файла на пять байтов от конца файла:

```
aFile.Seek(-5, SeekOrigin.End);
```

Файлы с возможностью доступа в такой манере иногда называют *файлами произвольного доступа*, потому что приложение может обращаться к любой позиции внутри файла. Классы Stream, описанные далее, обращаются к файлам последовательно и не позволяют манипулировать указателями файла подобным образом.

В версии .NET 4 появилось новое пространство имен по имени System.IO.MemoryMappedFiles, которое содержит типы (такие как MemoryMappedFile), предоставляющие альтернативные средства для произвольного доступа к очень большим файлам. Это пространство имен в настоящей главе не рассматривается, однако с ним стоит ознакомиться, если есть вероятность столкнуться с упомянутым сценарием.

Чтение данных

Чтение данных с использованием класса FileStream реализуется не так просто, как с помощью класса StreamReader, который будет описан в конце главы. Это связано с тем, что класс FileStream имеет дело исключительно с низкоуровневыми байтами. Работа с низкоуровневыми байтами делает класс FileStream полезным для любого рода файлов данных, а не только для текстовых. За счет чтения байтовых данных объект FileStream может применяться для чтения таких файлов, как файлы изображений или звуковые файлы. Платой за гибкость является невозможность использования FileStream для чтения данных непосредственно в строку, как это делается посредством класса StreamReader. Однако несколько классов преобразования позволяют довольно легко получать из байтовых массивов символьные и наоборот.

Метод FileStream.Read() — главное средство доступа к данным из файла, на который указывает объект FileStream. Этот метод читает данные из файла и затем пишет их в массив byte. Он принимает три параметра, первый из которых представляет массив byte для приема данных из объекта FileStream. Во втором параметре указывается позиция в этом массиве, куда нужно писать данные — обычно это 0, чтобы начать запись данных из файла в начало массива. Последний параметр задает количество байт для чтения из файла.

В следующем практическом занятии демонстрируется чтение данных из файла произвольного доступа. Файл, из которого вы будете читать — это в действительности файл класса, который создается для примера.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Чтение данных из файла произвольного доступа

1. Создайте новое консольное приложение по имени `ReadFile` и сохраните его в каталоге `C:\BegVCSharp\Chapter21`.
2. Добавьте следующую директиву `using` в начало файла `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

Фрагмент кода ***ReadFile\Program.cs***

3. Добавьте в метод `Main()` следующий код:

```
static void Main(string[] args)
{
  byte[] byData = new byte[200];
  char[] charData = new Char[200];
  try
  {
    FileStream aFile = new FileStream("../../Program.cs", FileMode.Open);
    aFile.Seek(113, SeekOrigin.Begin);
    aFile.Read(byData, 0, 200);
  }
  catch(IOException e)
  {
    Console.WriteLine("An IO exception has been thrown!");
    // Сгенерировано исключение ввода-вывода!
    Console.WriteLine(e.ToString());
    Console.ReadKey();
    return;
  }
  Decoder d = Encoding.UTF8.GetDecoder();
  d.GetChars(byData, 0, byData.Length, charData, 0);
  Console.WriteLine(charData);
  Console.ReadKey();
}
```

4. Запустите приложение. Результат показан на рис. 21.2.

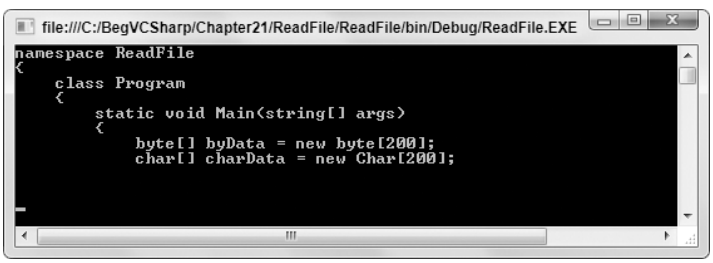


Рис. 21.2. *Результат работы приложения* `ReadFile`

Описание работы

Это приложение открывает файл `.cs` для чтения. Оно делает это, переходя на два каталога выше по файловой структуре с помощью строки `..` в следующем операторе:

```
FileStream aFile = new FileStream("../../Program.cs", FileMode.Open);
```

Два оператора, реализующих собственно поиск и чтение из определенной точки файла, выглядят так:

```
aFile.Seek(113, SeekOrigin.Begin);
aFile.Read(byData, 0, 200);
```

Первая строка перемещает указатель в файле на байт номер 113. Это будет буква n в namespace в файле Program.cs; предшествующие ей 113 символов — это директивы using. Вторая строка читает следующие 200 символов в байтовый массив byData.

Обратите внимание, что эти две строки помещены в блоки try...catch для обработки любых исключений, которые могут быть сгенерированы:

```
try
{
   aFile.Seek(135, SeekOrigin.Begin);
   aFile.Read(byData,0,100);
}
catch(IOException e)
try
{
   aFile.Seek(135, SeekOrigin.Begin);
   aFile.Read(byData,0,100);
}
catch(IOException e)
```

Почти все операции, включающие файловый ввод-вывод, могут сгенерировать исключение типа IOException. Любой производственный код должен содержать обработку ошибок, особенно при работе с файловой системой. Во всех примерах, приведенных в этой главе, включена базовая форма обработки ошибок.

После получения массива byte из файла потребуется преобразовать его в символьный массив, чтобы его можно было отобразить на консоли. Для этого воспользуйтесь классом Decoder из пространства имен System.Text. Этот класс спроектирован для преобразования низкоуровневых байтов в более подходящие элементы, такие как символы:

```
Decoder d = Encoding.UTF8.GetDecoder();
d.GetChars(byData, 0, byData.Length, charData, 0);
```

Эти строки создают объект Decoder на основе схемы кодирования UTF-8, которая представляет собой схему кодирования Unicode. Затем вызывается метод GetChars(), принимающий массив байт и преобразующий его в массив символов. После того, как это будет сделано, символьный массив может быть выведен на консоль.

Запись данных

Процесс записи в файл произвольного доступа очень похож. Сначала должен быть создан байтовый массив; простейший способ сделать это предусматривает сначала построение символьного массива, который планируется записать в файл. Затем с использованием объекта Encoder он должен быть преобразован в байтовый массив, почти так же, как применяется объект Decoder. И, наконец, нужно вызвать метод Write() для отправки массива в файл.

Рассмотрим простой пример, демонстрирующий, как это делается.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Запись данных в файл произвольного доступа**

1. Создайте консольное приложение по имени WriteFile и сохраните его в каталоге C:\BegVCSharp\Chapter21.

2. Добавьте следующую директиву `using` в файл `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

Фрагмент кода ***WriteFile\Program.cs***

3. Добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
  byte[] byData;
  char[] charData;
  try
  {
    FileStream aFile = new FileStream("Temp.txt", FileMode.Create);
    charData = "My pink half of the drainpipe.".ToCharArray();
    byData = new byte[charData.Length];
    Encoder e = Encoding.UTF8.GetEncoder();
    e.GetBytes(charData, 0, charData.Length, byData, 0, true);
    // Переместить файловый указатель в начало файла.
    aFile.Seek(0, SeekOrigin.Begin);
    aFile.Write(byData, 0, byData.Length);
  }
  catch (IOException ex)
  {
    Console.WriteLine("An IO exception has been thrown!"); // исключение
                                                           // ввода-вывода
    Console.WriteLine(ex.ToString());
    Console.ReadKey();
    return;
  }
}
```

4. Запустите приложение. Оно должно быстро выполниться и закрыться.
5. Перейдите в каталог приложения. Файл должен быть сохранен там, потому что применялся относительный путь — `WriteFile\bin\Debug`. Откройте файл `Temp.txt`. Вы должны увидеть в файле текст, как показано на рис. 21.3.

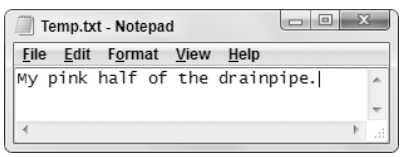


Рис. 21.3. *Результирующий файл* `Temp.txt`

Описание работы

Это приложение открывает файл в его собственном каталоге и записывает в него строку. По структуре этот пример очень похож на предыдущий, за исключением того, что используется `Write()` вместо `Read()` и `Encoder` вместо `Decoder`.

Следующая строка кода создает символьный массив с использованием статического метода `ToCharArray()` класса `String`. Поскольку все в C# является объектами, и текст `My pink half of the drainpipe.` — это на самом деле объект `string` (хотя и несколько странный), такие статические методы могут быть вызваны даже на строке символов:

```
CharData = "My pink half of the drainpipe.".ToCharArray();
```

В приведенных ниже строках показано, как преобразовать символьный массив в корректный байтовый массив, необходимый объекту FileStream:

```
Encoder e = Endoding.UTF8.GetEncoder();
e.GetBytes(charData, 0, charData.Length, byData, 0, true);
```

На этот раз объект Encoder создается на основе кодировки UTF-8. Кодировка Unicode также используется для декодирования, и на этот раз нужно закодировать символьные данные в корректный байтовый формат, прежде чем его можно будет записать в поток. Метод GetBytes() — место, где происходит вся магия. Он преобразует символьный массив в байтовый массив. GetBytes() принимает символьный массив в качестве первого параметра (charData в рассматриваемом примере) и индекс, указывающий на начало массива, в качестве второго (0 — для начала массива). Третий параметр — это количество символов, подлежащих преобразованию (charData.Length — количество элементов в массиве charData). Четвертый параметр — байтовый массив, в который нужно поместить данные (byData), а пятый параметр — индекс, указывающий на начало записи в байтовом массиве (0 — для начала массива byData).

Шестой, и последний, параметр определяет, должен ли объект Encoder сбрасывать свое состояние после завершения работы. Это отражает тот факт, что объект Encoder запоминает место в памяти, где он остановился в байтовом массиве. Это предназначено для последовательных вызовов объекта Encoder, но бессмысленно при единственном его вызове. Финальный вызов Encoder должен установить этот параметр в true, чтобы очистить его память и освободить объект для сборки мусора.

После этого остается лишь записать байтовый массив в FileStream с использованием метода Write():

```
aFile.Seek(0, SeekOrigin.Begin);
aFile.Write(byData, 0, byData.Length);
```

Подобно Read(), метод Write() принимает три параметра: массив, из которого производится запись, индекс в массиве, откуда будет начинаться запись, и количество записываемых байтов.

Объект StreamWriter

Работа с массивами байтов не приводит в восторг большинство людей; имея дело с объектом FileStream, вы можете предположить существование другого пути. И действительно: получив объект FileStream, вы обычно помещаете его в оболочку StreamWriter или StreamReader и используете его методы для манипулирования файлом. Если возможность переставлять файловый указатель в произвольную позицию не нужна, эти классы значительно облегчают работу с файлами.

Класс StreamWriter позволяет записывать символы и строки в файл, оставляя классу внутреннее преобразование и запись объекта FileStream.

Существует много способов создания объекта StreamWriter. Если объект FileStream уже имеется, для создания StreamWriter можно использовать следующий код:

```
FileStream aFile = new FileStream("Log.txt", FileMode.CreateNew);
StreamWriter sw = new StreamWriter(aFile);
```

Объект StreamWriter можно также создать непосредственно из файла:

```
StreamWriter sw = new StreamWriter("Log.txt", true);
```

Этот конструктор принимает имя файла и булевское значение, которое указывает, следует дописывать файл или создать новый.

- Если оно установлено в false, создается новый файл, либо же существующий файл усекается до нулевого размера, а затем открывается.

➤ Если оно установлено в `true`, файл открывается, и его данные сохраняются. Если файла нет, он создается.

В отличие от `FileStream`, создание `StreamWriter` не предоставляет аналогичного набора опций. Помимо булевского значения для дописывания или создания нового файла, нет возможности специфицировать свойство `FileMode`, как это делалось с классом `FileStream`. Не имея возможности установить свойство `FileAccess`, вы всегда имеете привилегии чтения/записи файла. Чтобы использовать любые из расширенных параметров, их сначала потребуется указать в конструкторе `FileStreeam`, а затем создать `StreamWriter` из объекта `FileStream`, как это продемонстрировано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Запись данных в выходной поток

1. Создайте консольное приложение по имени `StreamWriter` и сохраните его в каталоге `C:\BegVCSharp\Chapter21`.
2. Вы снова будете использовать пространство имен `System.IO`, поэтому добавьте следующую директиву `using` в начало файла `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

*Фрагмент кода **StreamWrite\Program.cs***

3. Добавьте в метод `Main()` следующий код:

```
static void Main(string[] args)
{
  try
  {
    FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate);
    StreamWriter sw = new StreamWriter(aFile);
    bool truth = true;

    // Записать данные в файл.
    sw.WriteLine("Hello to you.");
    sw.WriteLine("It is now {0} and things are looking good.",
    DateTime.Now.ToLongDateString());
    sw.Write("More than that,");
    sw.Write(" it's {0} that C# is fun.", truth);
    sw.Close();
  }
  catch(IOException e)
  {
    Console.WriteLine("An IO exception has been thrown!"); // исключение
                                                           // ввода-вывода
    Console.WriteLine(e.ToString());
    Console.ReadLine();
    return;
  }
}
```

4. Скомпилируйте и запустите проект. Если никаких ошибок не будет найдено, он должен быстро отработать и закрыться. Поскольку на консоли ничего не отображается, то и смотреть тут особо не на что.

5. Перейдите в каталог приложения и найдите файл `Log.txt`. Он расположен в папке `StreamWrite\bin\Debug`, так как указывался относительный путь.
6. Откройте файл. Вы должны увидеть то, что показано на рис. 21.4.

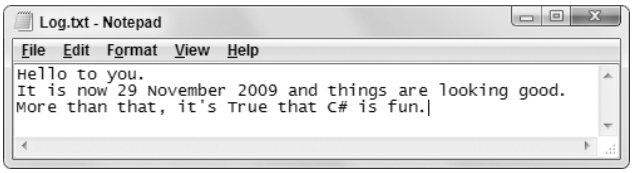


Рис. 21.4. *Результирующий файл* `Log.txt`

Описание работы

Это простое приложение демонстрирует два наиболее важных метода класса `StreamWriter` — `Write()` и `WriteLine()`. Оба они имеют множество перегруженных версий для выполнения более развитого файлового вывода, но в этом примере используется базовый вывод строк.

Метод `WriteLine()` пишет переданную строку, немедленно дополняя ее символом новой строки. В примере видно, что это заставляет следующую операцию записи начинаться с новой строки.

Точно так же, как вы пишете форматированный вывод на консоль, это можно делать и в файл. Например, вот как вывести значения переменных в файл, используя стандартные параметры формата:

```
sw.WriteLine("It is now {0} and things are looking good.",
             DateTime.Now.ToLongDateString());
```

`DateTime.Now` хранит текущую дату; метод `ToLongDateString()` позволяет преобразовать эту дату в легко читаемую форму.

Метод `Write()` просто пишет переданную строку в файл без добавления символа новой строки, позволяя записывать целые предложения или абзацы с использованием дополнительных операторов `Write()`:

```
sw.Write("More than that,");
sw.Write(" it's {0} that C# is fun.", truth);
```

Здесь, опять-таки, применяются параметры формата — на этот раз с вызовом `Write()`, — чтобы отобразить булевское значение “истина”; эта переменная ранее устанавливается в `true`, и ее значение при форматировании автоматически преобразуется в строку `True`.

Метод `Write()` и параметры формата можно использовать для записи разделенных запятыми полей:

```
[Объект_StreamWriter].Write("{0},{1},{2}", 100, "A nice product", 10.50);
```

В более сложном примере данные могли бы поступать из базы данных или другого источника.

Объект `StreamReader`

Входные потоки используются для чтения данных из внешних источников. Часто это файл на диске или где-то в сети, но помните, что источником может быть почти все что угодно, посылающее данные, например, сетевое приложение, веб-служба или даже консоль.

Класс `StreamReader` — это то, что вы будете применять для чтения данных из файлов. Подобно классу `StreamWriter`, это обобщенный класс, который может использоваться с любым потоком. В следующем практическом занятии снова конструируется объект `FileStream`, который будет указывать на корректный файл.

Объекты StreamReader создаются почти так же, как объекты StreamWriter. Наиболее распространенный способ его создания — применение ранее созданного объекта FileStream:

```
FileStream aFile = new FileStream("Log.txt", FileMode.Open);
StreamReader sr = new StreamReader(aFile);
```

Подобно StreamWriter, объект класса StreamReader может быть создан непосредственно из строки, содержащей путь к определенному файлу:

```
StreamReader sr = new StreamReader("Log.txt");
```

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Чтение данных из входного потока

1. Создайте новое консольное приложение по имени StreamRead и сохраните его в каталоге C:\BegVCSharp\Chapter21.
2. Импортируйте пространство имен System.IO, поместив следующую строку кода в начало файла Program.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

3. Добавьте следующий код в метод Main():

```
static void Main(string[] args)
{
  string line;
  try
  {
    FileStream aFile = new FileStream("Log.txt", FileMode.Open);
    StreamReader sr = new StreamReader(aFile);
    line = sr.ReadLine();
    // Прочитать данные строка за строкой.
    while(line != null)
    {
      Console.WriteLine(line);
      line = sr.ReadLine();
    }
    sr.Close();
  }
  catch(IOException e)
  {
    Console.WriteLine("An IO exception has been thrown!"); // исключение
                                                           // ввода-вывода
    Console.WriteLine(e.ToString());
    return;
  }
  Console.ReadKey();
}
```

4. Скопируйте файл Log.txt, созданный в предыдущем примере, в каталог StreamRead\bin\Debug. Если файла по имени Log.txt нет, конструктор FileStream сгенерирует исключение.
5. Запустите приложение. Вы должны увидеть выведенное на консоль содержимое файла, как показано на рис. 21.5.

Рис. 21.5. *Результат работы приложения* `StreamRead`

Описание работы

Это приложение очень похоже на предыдущее, но с очевидным отличием — на этот раз осуществляется чтение файла вместо записи в него. Как и ранее, вы должны импортировать пространство имен `System.IO`, чтобы иметь доступ к необходимым классам.

Для чтения текста из файла применяется метод `ReadLine()`. Метод читает текст до тех пор, пока не встретит символ новой строки, и возвращает прочитанный текст в виде строки. Метод возвращает `null`, когда достигнут конец файла, что можно использовать для проверки наступления этого условия. Обратите внимание на применение цикла `while`; это гарантирует, что прочитанная строка не будет равна `null`, прежде чем выполнится любой код в теле цикла — таким образом, будет отображено только реальное содержимое файла:

```
line = sr.ReadLine();
while( line != null)
{
  Console.WriteLine(strLine);
  line = sr.ReadLine();
}
```

Чтение данных

Метод `ReadLine()` — не единственный способ получить доступ к данным в файле. Класс `StreamReader` содержит множество методов для чтения данных.

Простейший из методов чтения — `Read()`. Он возвращает следующий символ из потока как положительное целое число или `-1`, если достигнут конец потока. Это значение может быть преобразовано в символ с использованием служебного класса `Convert`. В предыдущем примере основные части программы можно было бы переписать так, как показано ниже:

```
StreamReader sr = new StreamReader(aFile);
int nChar;
nChar = sr.Read();
while(nChar != -1)
{
  Console.Write(Convert.ToChar(nChar));
  nChar = sr.Read();
}
sr.Close();
```

При работе с файлами небольших размеров очень удобен метод `ReadToEnd()`. Он читает весь файл целиком и возвращает его содержимое в виде строки. В этом случае предыдущее приложение может быть упрощено следующим образом:

```
StreamReader sr = new StreamReader(aFile);
line = sr.ReadToEnd();
Console.WriteLine(line);
sr.Close();
```

Хотя это может показаться простым и удобным, будьте осторожны. Читая все данные в строковый объект, вы помещаете все данные файла в память. В зависимости от размера данных это может быть крайне нежелательно. Если данные очень велики, лучше оставить их в файле и обращаться к ним методами `StreamReader`.

Другой способ работы с большими файлами, появившийся в .NET 4, состоит в использовании статического метода `File.ReadLines()`. На самом деле есть несколько статических методов `File`, которые можно применять для упрощения чтения и записи данных, но этот особенно интересен тем, что возвращает коллекцию `IEnumerable<string>`. Выполняя итерацию по строкам из этой коллекции, можно читать файл строка за строкой. С использованием этого метода предыдущий пример можно переписать, как показано ниже:

```
foreach (string alternativeLine in File.ReadLines("Log.txt"))
   Console.WriteLine(alternativeLine);
```

Как видите, в .NET предусмотрено несколько разных способов для достижения одного и того же результата, а именно — чтения данных из файла. Выбирайте технику, которая наиболее подходит в конкретной ситуации.

Файлы с разделителями

Файлы с разделителями — это распространенная форма хранения данных, используемая во многих унаследованных системах. Если разрабатываемое приложение должно взаимодействовать с такой системой, вы часто будете встречаться с форматом данных с разделителями. Чаще других используется разделитель в виде запятой. Например, данные из электронных таблиц Excel, из базы данных Access или SQL Server могут быть экспортированы в файл с разделителем-запятой (comma-separated value — CVS).

Ранее уже было показано, как применять класс `StreamWriter` для записи таких файлов. Читать их также несложно. Возможно, вы помните из главы 5 метод `Split()` класса `String`, применяемый для преобразования строки в массив на основе заданного символа-разделителя. Если вы укажете в качестве такого разделителя запятую, он создаст строковый массив правильной размерности, содержащий все данные из исходной строки с разделителем-запятой.

В следующем практическом занятии показано, насколько это может оказаться полезным. В примере используются значения, разделенные запятыми, которые загружаются в объект `List<Dictionary<string, string>>`. Этот полезный пример носит достаточно общий характер, и вы можете счесть удобным применение такой техники в собственном приложении, если необходимо работать с разделенными запятыми значениями.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Работа со значениями, разделенными запятыми**

1. Создайте новое консольное приложение по имени `CommaValues` и сохраните его в каталоге `C:\BegVCSharp\Chapter21`.
2. Поместите следующую строку в код в начало `Proram.cs`. Для работы с файлами нужно импортировать пространство имен `System.IO`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

*Фрагмент кода **CommaValues\Program.cs***

3. Добавьте следующий метод `GetData()` в тело `Program.cs` перед методом `Main()`:

```
private static List<Dictionary<string, string>> GetData(
   out List<string> columns)
{
  string line;
  string[] stringArray;
```

```
  char[] charArray = new char[] {','};
  List<Dictionary<string, string>> data = new List<Dictionary<string, string>>();
  columns = new List<string>();
  try
  {
    FileStream aFile = new FileStream(@"..\..\SomeData.txt", FileMode.Open);
    StreamReader sr = new StreamReader(aFile);
    // Получить столбцы из первой строки.
    // Разделить строку данных в строковый массив.
    line = sr.ReadLine();
    stringArray = strLine.Split(charArray);
    for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
    {
      columns.Add(stringArray[x]);
    }
    line = sr.ReadLine();
    while (line != null)
    {
      // Разделить строку данных в строковый массив.
      stringArray = line.Split(charArray);
      Dictionary<string, string> dataRow = new Dictionary<string, string>();
      for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
      {
        dataRow.Add(columns[x], stringArray[x]);
      }
      data.Add(dataRow);
      line = sr.ReadLine();
    }
    sr.Close();
    return data;
  }
  catch (IOException ex)
  {
    Console.WriteLine("An IO exception has been thrown!"); // исключение
                                                           //ввода-вывода
    Console.WriteLine(ex.ToString());
    Console.ReadLine();
    return data;
  }
}
```

4. Добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
  List<string> columns;
  List<Dictionary<string, string>> myData = GetData(out columns);
  foreach (string column in columns)
  {
    Console.Write("{0,-20}", column);
  }
  Console.WriteLine();
  foreach (Dictionary<string, string> row in myData)
  {
    foreach (string column in columns)
    {
      Console.Write("{0,-20}", row[column]);
    }
    Console.WriteLine();
  }
  Console.ReadKey();
}
```

5. Добавьте новый текстовый файл по имени `SomeData.txt`, выбрав элемент Text File (Текстовый файл) в диалоговом окне, доступном через пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент).

6. Введите следующий текст в новый текстовый файл:

```
ProductID,Name,Price
1,Spiky Pung,1000
2,Gloop Galloop Soup,25
4,Hat Sauce,12
```

Фрагмент кода CommaValues\SomeDate.txt

7. Запустите приложение. Вы должны увидеть текст файла, выведенный на консоль, как показано на рис. 21.6.

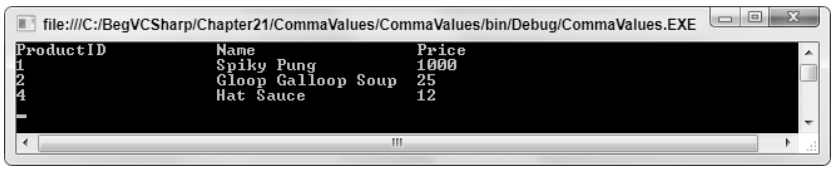


Рис. 21.6. *Результат работы приложения* `CommaValues`

Описание работы

Как и в предыдущем примере, это приложение читает файл строку за строкой в объект `string`. Однако поскольку известно, что этот файл содержит разделенные запятыми текстовые значения, они обрабатываются по-другому. Кроме того, в действительности прочитанные значения будут сохраняться в структуру данных.

Сначала нужно взглянуть на сами разделенные запятыми данные:

```
ProductID,Name,Price
1,Spiky Pung,1000
```

Первая строка содержит имена столбцов данных, а следующие за ней строки — сами данные. Таким образом, вы должны получить имена столбцов из первой строки файла и затем извлечь данные из остальных строк.

Метод `GetData()` объявлен статическим, поэтому его можно вызвать без создания экземпляра класса. Этот метод возвращает объект `List<Dictionary<string, string>>`, который вы создадите и затем наполните данными из разделенного запятыми текстового файла. Он также вернет объект `List<string>`, содержащий имена заголовков. Следующий код инициализирует эти объекты:

```
List<Dictionary<string, string>> data = new List<Dictionary<string, string>>();
columns = new List<string>();
```

`columns` будет содержать имена столбцов, взятые из первой строки текстового файла, разделенной запятыми, а `data` — значения последующих строк файла.

Все начинается с создания объекта `FileStream`, а затем вокруг него конструируется `StreamReader`, как это делалось в предыдущих примерах. Теперь можно прочитать первую строку файла и создать массив строк из этой одной строки:

```
line = sr.ReadLine();
stringArray = strLine.Split(charArray);
```

Метод `Split()`, который рассматривался в главе 5, принимает символьный массив (в данном случае содержащий только `','`), так что `stringArray` будет хранить массив строк, сформированный путем разбиения `strLine` по экземплярам `','`. Поскольку в на-

стоящий момент осуществляется чтение из первой строки файла, и эта строка содержит имена столбцов данных, необходимо пройтись по каждой строке в stringArray и добавить ее в columns:

```
for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
{
   columns.Add(stringArray[x]);
}
```

Теперь, имея имена столбцов данных, можно приступить к чтению самих данных. Код, необходимый для этого, по существу тот же, что и в ранее приведенном примере StreamRead, за исключением присутствия кода, нужного для добавления к data объекта Dictionary<string, string>:

```
line = sr.ReadLine();
while (line != null)
{
  // Разбиение строки данных на строковый массив.
  stringArray = strLine.Split(charArray);
  Dictionary<string, string> dataRow = new Dictionary<string, string>();
  for (int x = 0; x <= stringArray.GetUpperBound(0); x++)
  {
    dataRow.Add(columns[x], stringArray[x]);
  }
  data.Add(dataRow);
  line = sr.ReadLine();
}
```

Для каждой строки файла создается новый объект Dictionary<string, string>, который заполняется элементами этой строки. Каждый элемент этой коллекции имеет ключ, соответствующий имени столбца, и значение, представляющее собой значение столбца этой строки. Ключи извлекаются из созданного ранее объекта columns и значений, поступающими из строкового массива, который получен из Split() для строки текста, извлеченной из файла данных.

После чтения всех данных из файла StreamReader закрывается, а данные возвращаются. Код в методе Main() получает данные из метода GetData() в переменные myData и columns и отображает информацию на консоли. Сначала выводятся имена столбцов:

```
foreach (string column in columns)
{
   Console.Write("{0,-20}", column);
}
Console.WriteLine();
```

Часть -20 форматной строки {0,-20} гарантирует, что отображаемое имя будет выровнено влево в столбце шириной 20 символов; это поможет форматировать вывод.

И, наконец, осуществляется проход в цикле по объекту Dictionary<string, string> в коллекции myData и отображение значений в строке, каждый раз с использованием форматирующей строки для вывода:

```
foreach (Dictionary<string, string> row in myData)
{
   foreach (string column in columns)
   {
      Console.Write("{0,-20}", row[column]);
   }
   Console.WriteLine();
}
```

Как видите, извлечь осмысленные данные из файла со значениями, разделенными запятыми (CVS), с помощью .NET Framework достаточно просто. Эту технику также легко ком-

бинировать с приемами доступа к данным, которые будут показаны в последующих главах, и это значит, что данными, полученными из файла CSV, можно манипулировать так же, как и любыми другими источниками данных (наподобие баз данных). Однако из файла CSV не извлекается никакой информации о типах данных. В настоящее время все данные трактуются как строки. Для бизнес-приложения масштаба предприятия следует предусмотреть дополнительный шаг добавления информации о типе к извлекаемым данным. В зависимости от конкретного приложения, это может поступать из дополнительной информации, хранимой в файле CSV, конфигурироваться вручную или же выводиться из строк файла.

Несмотря на то что формат XML, рассматриваемый в следующей главе — это превосходный метод хранения и транспортировки данных, файлы CSV все еще достаточно распространены и останутся таковыми еще долгое время. Файлы, хранящие данные, разделенные запятыми, весьма лаконичны, и потому они меньше своих XML-аналогов.

Чтение и запись сжатых файлов

Часто при работе с файлами требуется значительное пространство на жестком диске. Это особенно верно для графических и звуковых файлов. Скорее всего, вам встречались утилиты, позволяющие сжимать и распаковывать файлы, что удобно для их передачи по электронной почте. Пространство имен `System.IO.Compression` содержит классы, позволяющие сжимать файлы из кода с использованием алгоритма GZIP или Deflate — оба они общедоступны для любого применения.

Однако при работе со сжатыми файлами есть еще кое-что помимо собственно сжатия. Коммерческие приложения позволяют множеству файлов размещаться в единственном сжатом файле и т.п. То, что вы увидите в настоящем разделе, много проще: сохранение текстовых данных в сжатом файле. Вряд ли удастся манипулировать этим файлом с помощью внешней утилиты, но этот файл будет намного меньше, чем его несжатый эквивалент!

Два класса сжимающих потоков из пространства имен `System.IO.Compression`, которые здесь рассматриваются — `DeflateStream` и `GZipStream` — работают очень похожим образом. В обоих случаях они инициализируются существующим потоком, которым в случае файлов является объект `FileStream`. После этого их можно применять вместе с `StreamReader` и `StreamWriter`, как и любой другой поток. Все, что потребуется специфицировать в дополнение — будет поток использоваться для сжатия (сохранения файлов) или распаковки (загрузки файлов), чтобы класс знал, что делать с переданными ему данными. Хорошая иллюстрация предлагается в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Чтение и запись сжатых данных

1. Создайте новое консольное приложение по имени `Compressor` и сохраните его в каталоге `C:\BegVCSharp\Chapter21`.
2. Поместите следующие строки кода в начало `Program.cs`. Вы должны импортировать пространство имен `System.IO` для работы с файлами, и `System.IO.Compression` для использования сжимающих классов:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.IO.Compression;
```

Фрагмент кода ***Compressor/Program.cs***

3. Добавьте следующие методы в тело `Program.cs` перед методом `Main()`:

```
static void SaveCompressedFile(string filename, string data)
{
  FileStream fileStream =
    new FileStream(filename, FileMode.Create, FileAccess.Write);
  GZipStream compressionStream =
    new GZipStream(fileStream, CompressionMode.Compress);
  StreamWriter writer = new StreamWriter(compressionStream);
  writer.Write(data);
  writer.Close();
}
static string LoadCompressedFile(string filename)
{
  FileStream fileStream =
    new FileStream(filename, FileMode.Open, FileAccess.Read);
  GZipStream compressionStream =
    new GZipStream(fileStream, CompressionMode.Decompress);
  StreamReader reader = new StreamReader(compressionStream);
  string data = reader.ReadToEnd();
  reader.Close();
  return data;
}
```

4. Добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
  try
  {
    string filename = "compressedFile.txt";
    Console.WriteLine(
      "Enter a string to compress (will be repeated 100 times):");
    // Console.WriteLine(
    //   "Введите строку для сжатия (она будет повторена 100 раз):");
    string sourceString = Console.ReadLine();
    StringBuilder sourceStringMultiplier =
      new StringBuilder(sourceString.Length * 100);
    for (int i = 0; i < 100; i++)
    {
      sourceStringMultiplier.Append(sourceString);
    }
    sourceString = sourceStringMultiplier.ToString();
    Console.WriteLine("Source data is {0} bytes long.", sourceString.Length);
    // Console.WriteLine(
    //   "Исходные данные имеют длину {0} байт.", sourceString.Length);
    SaveCompressedFile(filename, sourceString);
    Console.WriteLine("\nData saved to {0}.", filename);
    // Console.WriteLine("\nДаные сохранены в {0}.", filename);
    FileInfo compressedFileData = new FileInfo(filename);
    Console.WriteLine("Compressed file is {0} bytes long.",
                      compressedFileData.Length);
    // Console.WriteLine("Сжатый файл имеют длину {0} байт.",
    //                   compressedFileData.Length);
    string recoveredString = LoadCompressedFile(filename);
    recoveredString = recoveredString.Substring(0, recoveredString.Length / 100);
    Console.WriteLine("\nRecovered data: {0}", recoveredString);
    // Console.WriteLine("\nИзвлеченные данные: {0}", recoveredString);
    Console.ReadKey();
  }
  catch (IOException ex)
  {
```

```
      Console.WriteLine("An IO exception has been thrown!"); // исключение
                                                             // ввода-вывода
      Console.WriteLine(ex.ToString());
      Console.ReadKey();
    }
  }
```

5. Запустите приложение и введите достаточно длинную строку. Пример результата показан на рис. 21.7.

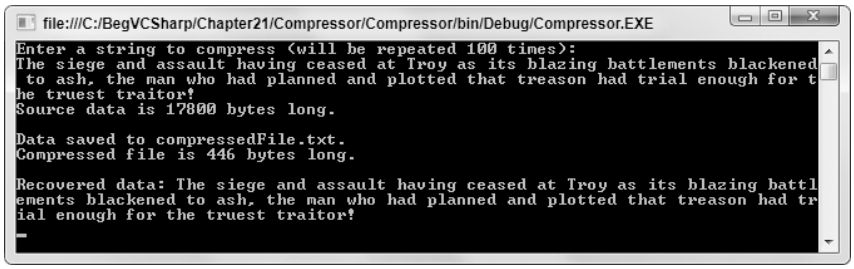


Рис. 21.7. *Результат работы приложения* `Compressor`

6. Откройте файл `compressedFile.txt` в редакторе Notepad. Содержимое этого файла показано на рис. 21.8.

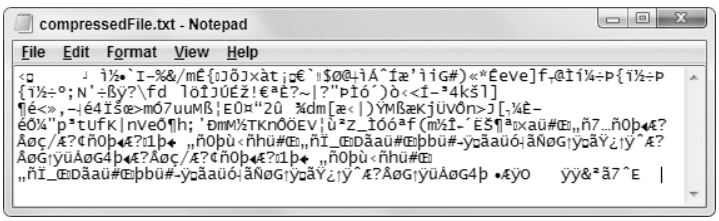


Рис. 21.8. *Содержимое файла* `compressedFile.txt`

Описание работы

В этом примере были определены два метода для сохранения и загрузки сжатого текстового файла. Первый из них — `SaveCompressedFile()` — выглядит так:

```
static void SaveCompressedFile(string filename, string data)
{
  FileStream fileStream =
    new FileStream(filename, FileMode.Create, FileAccess.Write);
  GZipStream compressionStream =
    new GZipStream(fileStream, CompressionMode.Compress);
  StreamWriter writer = new StreamWriter(compressionStream);
  writer.Write(data);
  writer.Close();
}
```

Код начинается с создания объекта `FileStream` и затем использует его для создания объекта `GZipStream`. Обратите внимание, что в приведенном коде все вхождения `GZipStream` можно было бы заменить `DefalateStream` — эти классы работают одинаково. Перечисление `CompressionMode.Compress` позволяет указать, что данные должны быть сжаты. После этого применяется `StreamWriter` для записи данных в файл.

Метод LoadCompressedFile() — зеркальное отображение метода SaveCompressedFile(). Вместо сохранения в файл с заданным именем он загружает сжатый файл в строку:

```
static string LoadCompressedFile(string filename)
{
  FileStream fileStream =
    new FileStream(filename, FileMode.Open, FileAccess.Read);
  GZipStream compressionStream =
    new GZipStream(fileStream, CompressionMode.Decompress);
  StreamReader reader = new StreamReader(compressionStream);
  string data = reader.ReadToEnd();
  reader.Close();
  return data;
}
```

Как и следовало ожидать, отличие связано с разными значениями перечислений FileMode, FileAccess и CompressionMode для загрузки и распаковки данных, а также использовании StreamReader для получения распакованного текста из файла.

Код в Main() предназначен просто для тестирования этих методов. Он запрашивает строку, дублирует ее 100 раз, чтобы сделать все интереснее, сжимает в файл, а затем извлекает все это оттуда. В рассмотренном примере начальная строфа из "Рыцарей круглого стола", повторенная 100 раз, имеет длину в 17 800 символов, но в сжатом виде состоит всего лишь из 441 байт, что составляет степень сжатия 40:1. Правда, здесь присутствует элемент мошенничества — известно, что алгоритм GZIP особенно эффективно работает с повторяющимися данными, однако это иллюстрирует сжатие в действии.

Вы также видели текст, хранимый в сжатом файле. Очевидно, что он нечитабелен, и это надо учитывать, если планируется разделять данные между приложениями. Однако поскольку файл был сжат известным алгоритмом, по крайней мере, другие приложения могут распаковывать его.

Сериализованные объекты

Как известно, приложения часто нуждаются в хранении данных на жестком диске. До сих пор в этой главе вы видели конструирование текста и файлов данных порция за порцией, но часто такой способ не совсем удобен. Иногда лучше сохранять данные в той форме, в которой они используются, а именно — в виде объектов.

В .NET Framework предлагается инфраструктура для сериализации объектов в пространствах имен System.Runtime.Serialization и System.Runtime.Srialization.Formatters, со специфическими классами, реализующими эту инфраструктуру в пространствах имен, которые вложены в упомянутые два. Доступны две реализации:

1. System.Runtime.Serialization.Formatters.Binary — это пространство имен содержит класс BinaryFormatter, который способен сериализовать объекты в двоичные данные и обратно;
2. System.Runtime.Serialization.Formatters.Soap — это пространство имен содержит класс SoapFormatter, который способен сериализовать объекты в XML-данные SOAP-формата и наоборот.

В этой главе рассматривается только BinaryFormatter, т.к. до XML пока дело не дошло. На самом деле применять форматировщик SoapFormatter не рекомендуется, хотя иногда это удобно, когда нужна читабельная для человека сериализация. Однако поскольку эти классы реализуют интерфейс IFormatter, большая часть последующего обсуждения касается обоих классов.

Интерфейс `IFormatter` также реализован двумя другими классами .NET Framework. Первый из них, `ObjectStateFormatter`, используется в ASP.NET для сериализации визуального состояния. Другой — `NetDataCOntract Serializer` — применяется для сериализации контрактов данных WCF.

Интерфейс `IFormatter` предоставляет методы, перечисленные в табл. 21.9.

Таблица 21.9. Методы интерфейса `IFormatter`

Метод	Описание
`void Serialize( Stream stream, object source)`	Сериализует объект `source` в поток `stream`
`object Deserialize( Stream stream)`	Десериализует данные в `stream` и возвращает результирующий объект

Важным и удобным для настоящей главы является то, что эти методы работают с потоками. Это облегчает их подключение к механизмам доступа к файлам, уже показанным в главе — можно легко использовать объекты `FileStream`.

Сериализация с применением `BinaryFormatter` проста:

```
IFormatter serializer = new BinaryFormatter();
serializer.Serialize(myStream, myObject);
```

Десериализация не сложнее:

```
IFormatter serializer = new BinaryFormatter();
MyObjectType myNewObject = serializer.Deserialize(myStream) as MyObjectType;
```

Очевидно, что нужны потоки и объекты, с которыми можно работать, но приведенный выше синтаксис подходит в большинстве случаев. В следующем практическом занятии показано, как это реально работает.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Сериализация и десериализация объектов

1. Создайте новое консольное приложение по имени `ObjectStore` и сохраните его в каталоге `C:\BegVCSharp\Chapter21`.
2. Добавьте к проекту новый класс по имени `Product` и модифицируйте код следующим образом:

```
namespace ObjectStore
{
  public class Product
  {
    public long Id;
    public string Name;
    public double Price;
    [NonSerialized]
    string Notes;

    public Product(long id, string name, double price, string notes)
    {
      Id = id;
      Name = name;
      Price = price;
      Notes = notes;
    }
```

```
    public override string ToString()
    {
      return string.Format("{0}: {1} (${2:F2}) {3}", Id, Name, Price, Notes);
    }
  }
}
```

Фрагмент кода ***ObjectStore\Product.cs***

3. Поместите следующие строки кода в начало `Program.cs`. Нужно импортировать пространство имен `System.IO` для работы с файлами, а другие пространства имен — для сериализации:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
```

Фрагмент кода ***ObjectStore\Program.cs***

4. Добавьте следующий код в метод `Main()` в `Program.cs`:

```
static void Main(string[] args)
{
  try
  {
    // Создать продукт.
    List<Product> products = new List<Product>();
    products.Add(new Product(1, "Spiky Pung", 1000.0, "Good stuff."));
    products.Add(new Product(2, "Gloop Galloop Soup", 25.0, "Tasty."));
    products.Add(new Product(4, "Hat Sauce", 12.0, "One for the kids."));
    Console.WriteLine("Products to save:");
    // Console.WriteLine("Продукты для сохранения:");
    foreach (Product product in products)
    {
      Console.WriteLine(product);
    }
    Console.WriteLine();
    // Получить сериализатор.
    IFormatter serializer = new BinaryFormatter();
    // Сериализовать продукты.
    FileStream saveFile =
      new FileStream("Products.bin", FileMode.Create, FileAccess.Write);
    serializer.Serialize(saveFile, products);
    saveFile.Close();
    // Десериализовать продукты.
    FileStream loadFile =
      new FileStream("Products.bin", FileMode.Open, FileAccess.Read);
    List<Product> savedProducts =
      serializer.Deserialize(loadFile) as List<Product>;
    loadFile.Close();
    Console.WriteLine("Products loaded:");
    // Console.WriteLine("Загруженные продукты:");
    foreach (Product product in savedProducts)
    {
      Console.WriteLine(product);
    }
  }
```

```
    catch (SerializationException e)
    {
      Console.WriteLine("A serialization exception has been thrown!");
      // Console.WriteLine("Сгенерировано исключение сериализации!");
      Console.WriteLine(e.Message);
    }
    catch (IOException e)
    {
      Console.WriteLine("An IO exception has been thrown!"); // исключение
                                                             //ввода-вывода
      Console.WriteLine(e.ToString());
    }
    Console.ReadKey();
  }
```

5. Запустите приложение. Результат должен получиться подобным показанному на рис. 21.9.

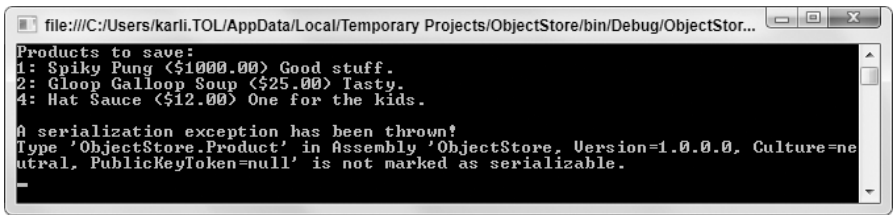


Рис. 21.9. *Во время работы приложения* `ObjectStore` *возникло исключение*

6. Модифицируйте код `Product.cs` следующим образом:

```
namespace ObjectStore
{
[Serializable]
public class Product
{
  ...
}
```

Фрагмент кода ***ObjectStore\Product.cs***

7. Запустите приложение снова. Результат представлен на рис. 21.10

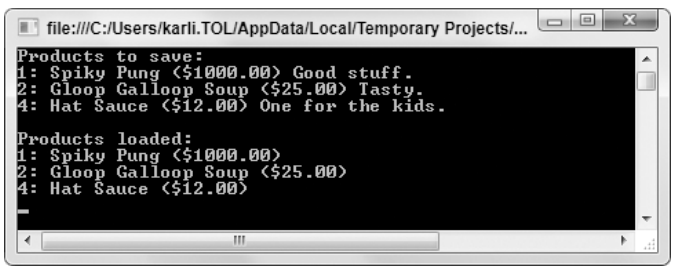


Рис. 21.10*. Результат работы приложения* `ObjectStore`

8. Откройте файл `Products.bin` в редакторе Notepad. Содержимое этого файла показано на рис. 21.11.

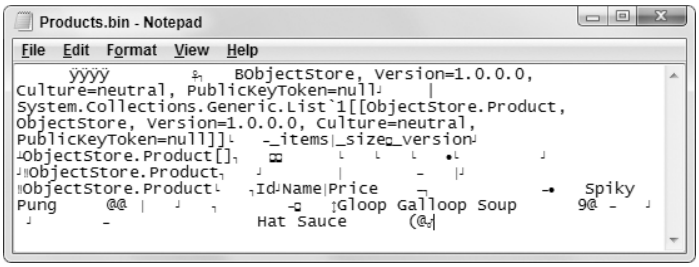


***Рис. 21.11**. Содержимое файла* `Products.bin`

Описание работы

В этом примере создается коллекция объектов `Product`, которая сохраняется на диске, а затем перезагружается оттуда. Однако при первом запуске приложения было сгенерировано исключение, потому что объект `Product` не был помечен как *сериализуемый*.

.NET Framework требует помечать объекты как сериализуемые, чтобы можно было выполнять их сериализацию. На то имеется несколько причин, включая перечисленные ниже.

- Некоторые объекты не очень хорошо сериализуются. Они могут требовать ссылок на локальные данные, которые, например, существуют только до тех пор, пока находятся в памяти.
- Некоторые объекты могут содержать важные данные, которые не должны сохраняться незащищенным способом или передаваться другому процессу.

Как показано в примере, пометить объект как сериализуемый очень просто — для этого применяется атрибут `Serializable`:

```
namespace ObjectStore
{
  [Serializable]
  public class Product
  {
    ...
  }
```

Обратите внимание, что этот атрибут не наследуется производными классами. Он должен быть применен к каждому классу, который планируется подвергать сериализации. Также следует отметить, что класс `List<T>`, использованный для генерации коллекции объектов `Product`, имеет этот атрибут; в противном случае применение его к `Product` не позволит сделать коллекцию сериализуемой.

Когда коллекция `products` успешно сериализована и десериализована (со второй попытки), становится очевидным другой важный факт. При десериализации восстанавливаются только поля `Id`, `Name` и `Price`. Это связано с использованием другого атрибута, а именно — `NonSearialized`:

```
[NonSerialized]
string Notes;
```

Любой член класса может быть помечен этим атрибутом, и тогда он не будет сохраняться вместе с другими членами. Это может быть полезно, если, например, только одно поле или свойство содержит ответственные данные.

Вы также видели результирующие сохраненные данные в этом примере. Некоторые данные здесь читабельны для человека; это может быть нежелательно или неожиданно. В классе `BinaryFormatter` не предпринимается серьезных попыток защитить данные от посто-

ронних глаз. Конечно, поскольку используются потоки, относительно легко перехватить данные при сохранении их на диске или загрузке приложением и применить собственный маскирующий или шифрующий алгоритм. То же касается сжатия — применяя технику из предыдущего раздела, довольно легко сжать данные объекта при сохранении на диск.

Тема сериализации включает еще многое, но изложенной информации достаточно, чтобы заложить основу. Одной из наиболее развитых техник, которые имеет смысл изучить, является специальная сериализация с применением интерфейса `ISerializable`, который позволяет точно настраивать, какие именно данные подлежат сериализации. Это может оказаться важным, например, при обновлении классов в последующем выпуске. Изменение списка членов, подлежащих сериализации, может привести к тому, что сохраненные предыдущей версией данные окажутся нечитабельными, если только вы не предусмотрите собственную логику сохранения и извлечения данных.

Мониторинг файловой системы

Иногда приложения должны делать нечто большее, чем просто чтение и запись файлов в рамках файловой системы. Например, может быть важно знать, когда модифицируются файлы или каталоги. .NET Framework облегчает задачу создания таких приложений, которые делают это.

Класс, помогающий в этом, называется `FileStreamWatcher`. Он предоставляет несколько событий, которые приложение может перехватить. В результате приложение может реагировать на события файловой системы.

Базовая процедура использования `FileStreamWatcher` проста. Сначала вы должны установить несколько свойств, которые укажут, где нужно выполнять мониторинг, что отслеживать и когда должно возникнуть событие, которое приложение обработает. Затем вы предоставляете ему адреса обработчиков событий, чтобы они вызывались при наступлении какого-то существенного события. И, наконец, вы запускаете его и ожидаете события.

Свойства, которые должны быть установлены перед включением объекта `FileStreamWatcher` в работу, перечислены в табл. 21.10.

Таблица 21.10. Свойства, которые должны быть установлены для объекта `FileStreamWatcher`

Свойство	Описание
`Path`	Должно быть установлено в местоположение или каталог, который нужно отслеживать
`NotifyFilter`	Комбинация значений перечисления `NotifyFilters`, указывающих, что именно нужно отслеживать в файлах, подверженных мониторингу. Это представляет свойства файла или папки, подверженной мониторингу. Если любое из указанных свойств изменяется, инициируется событие. Возможные значения перечисления — `Attributes`, `CreationTime`, `DirectoryName`, `FileName`, `LastAccess`, `LastWrite`, `Security` и `Size`. Обратите внимание, что их можно комбинировать с помощью бинарной операции “ИЛИ”
`Filter`	Фильтр, задающий файлы, которые нужно подвергать мониторингу, например, `*.txt`

Установив все это, вы должны написать обработчики четырех событий: `Changed`, `Created`, `Deleted` и `Renamed`. Как было показано в главе 13, для этого нужно всего лишь разработать собственный метод и назначить его событию объекта. Создавая собственные обработчики событий, вы обеспечиваете возможность их вызова при возникновении со-

бытия. Каждое событие инициируется, когда модифицируется файл или каталог, соответствующий `Path`, `NotifyFilter` и `Filter`.

Установив названные свойства и события, присвойте свойству `EnableRaisingEvents` значение `true`, чтобы запустить мониторинг. В следующем практическом занятии применяется `FileSystemWatcher` в простом клиентском приложении для слежения за выбранным каталогом.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Мониторинг файловой системы

1. Создайте новое консольное приложение по имени `FileWatch` и сохраните его в каталоге `C:\BegVCSharp\Chapter21`.
2. Установите различные свойства формы, как показано в следующей таблице:

Свойство	Установки
`FormBorderStyle`	`FixedDialog`
`MaximizeBox`	`False`
`MinimizeBox`	`False`
`Size`	`302, 160`
`StartPosition`	`CenterScreen`
`Text`	`File Monitor`

3. Добавьте на форму необходимые элементы управления и установите их свойства, как показано ниже:

Элемент управления	Name	Location	Size	Text
`TextBox`	`txtLocation`	`8, 26`	`184, 20`	
`Button`	`cmdBrowse`	`208, 24`	`64, 24`	`Browse...`
`Button`	`cmdWatch`	`88, 56`	`80, 32`	`Watch!`
`Label`	`lblWatch`	`8, 104`	`0, 13`	

Удостоверьтесь, что свойство `Enabled` кнопки `cmdWatch` установлено в `False`, поскольку нельзя отслеживать файл до того, как он будет указан, и установите свойство `AutoSize` метки `lblWatch` в `True`, чтобы видеть ее содержимое. Кроме того, добавьте на форму элемент управления `OpenDialog`, установив его свойство `Name` в `FileDialog`, а свойство `Filter` — в `All Files|*.*`. В результате должна получиться форма, подобная показанной на рис. 21.12.

Рис. 21.12. *Результирующая форма*

4. Теперь, когда форма построена, можно добавить код для выполнения некоторой работы. Для начала добавьте к существующему списку директив обычную директиву `using` для пространства имен `System.IO`:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
```

Фрагмент кода ***FileWatch\Form1.cs***

5. Добавьте класс `FileSystemWatcher` к классу `Form1`, а также делегат для облегчения изменения текста `lblWatch` из разных потоков. Поместите в `Form1.cs` следующий код:

```
namespace FileWatch
{
  partial class Form1 : Form
  {
    // Объект наблюдения за файловой системой.
    private FileSystemWatcher watcher;
    private delegate void UpdateWatchTextDelegate(string newText);
```

6. Сразу после вызова метода `InitializeComponent()` добавьте приведенный ниже код. Этот код необходим для инициализации объекта `FileStreamWatcher` и связывания событий с методами, которые будут созданы впоследствии.

```
public Form1()
{
  InitializeComponent();
  this.watcher = new System.IO.FileSystemWatcher();
  this.watcher.Deleted +=
    new System.IO.FileSystemEventHandler(this.OnDelete);
  this.watcher.Renamed +=
    new System.IO.RenamedEventHandler(this.OnRenamed);
  this.watcher.Changed +=
    new System.IO.FileSystemEventHandler(this.OnChanged);
  this.watcher.Created +=
    new System.IO.FileSystemEventHandler(this.OnCreate);
}
```

7. Добавьте в класс `Form1` показанные далее пять методов. Первый метод используется для асинхронного обновления текста в `lblWatch` из потоков, которые запустят обработчики событий `FileSystemWatcher`. Другие методы представляют собой сами обработчики событий.

```
// Служебный метод для обновления текста наблюдения.
public void UpdateWatchText(string newText)
{
  lblWatch.Text = newText;
}
// Определение обработчиков событий.
public void OnChanged(object source, FileSystemEventArgs e)
{
  try
  {
    StreamWriter sw =
      new StreamWriter("C:/FileLogs/Log.txt", true);
    sw.WriteLine("File: {0} {1}", e.FullPath,
                 e.ChangeType.ToString());
    sw.Close();
```

```
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Wrote change event to log"); // Запись в журнал события изменения
  }
  catch (IOException)
  {
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Error Writing to log"); // Ошибка при записи в журнал
  }
}
public void OnRenamed(object source, RenamedEventArgs e)
{
  try
  {
    StreamWriter sw =
      new StreamWriter("C:/FileLogs/Log.txt", true);
    sw.WriteLine("File renamed from {0} to {1}", e.OldName,
              e.FullPath); // Файл переименован
    sw.Close();
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Wrote renamed event to log"); // Запись в журнал события переименования
  }
  catch (IOException)
  {
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Error Writing to log"); // Ошибка при записи в журнал
  }
}
public void OnDelete(object source, FileSystemEventArgs e)
{
  try
  {
    StreamWriter sw =
      new StreamWriter("C:/FileLogs/Log.txt", true);
    sw.WriteLine("File: {0} Deleted", e.FullPath); // Файл удален
    sw.Close();
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Wrote delete event to log"); // Запись в журнал события удаления
  }
  catch (IOException)
  {
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Error Writing to log"); // Ошибка при записи в журнал
  }
}
public void OnCreate(object source, FileSystemEventArgs e)
{
  try
  {
    StreamWriter sw =
      new StreamWriter("C:/FileLogs/Log.txt", true);
    sw.WriteLine("File: {0} Created", e.FullPath); // Файл создан
    sw.Close();
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Wrote create event to log"); // Запись в журнал события создания
  }
  catch (IOException)
  {
    this.BeginInvoke(new UpdateWatchTextDelegate(UpdateWatchText),
      "Error Writing to log"); // Ошибка при записи в журнал
  }
}
```

8. Добавьте обработчик событий Click для кнопки Browse (Обзор). Код в этом обработчике событий открывает диалог Open File (Открытие файла), позволяя пользователю выбирать файл для мониторинга. Дважды щелкните на кнопке Browse и введите следующий код:

```
private void cmdBrowse_Click(object sender, EventArgs e)
{
  if (FileDialog.ShowDialog() != DialogResult.Cancel )
  {
    txtLocation.Text = FileDialog.FileName;
    cmdWatch.Enabled = true;
  }
}
```

Метод ShowDielog() возвращает значение перечисления DialogResult, отражающее способ выхода пользователя из диалога File Open. (Пользователь может щелкнуть на кнопке OK или Cancel (Отмена).) Необходимо удостовериться, что пользователь не щелкнул на кнопке Cancel, поэтому перед сохранением выбора пользователя в TextBox результат вызова метода сравнивается со значением перечисления DialogResult.Cancel. И, наконец, свойство Enabled кнопки Watch устанавливается в true, чтобы можно было отслеживать файл.

9. Двойной щелкните на кнопке Watch (Отслеживание) и добавьте следующий код обработчика событий Click для запуска FileSystemWatcher:

```
private void cmdWatch_Click(object sender, EventArgs e)
{
  watcher.Path =Path.GetDirectoryName(txtLocation.Text);
  watcher.Filter = Path.GetFileName(txtLocation.Text);
  watcher.NotifyFilter = NotifyFilters.LastWrite |
    NotifyFilters.FileName | NotifyFilters.Size;
  lblWatch.Text = "Watching " + txtLocation.Text;
  // Начать наблюдение.
  watcher.EnableRaisingEvents = true;
}
```

10. Удостоверьтесь, что каталог FileLogs существует, чтобы можно было записывать в него данные. Добавьте в конструктор Form1 следующий код, который проверит существование каталога и создаст его в случае его отсутствия:

```
public Form1()
{
  ...
  DirectoryInfo aDir = new DirectoryInfo(@"C:\FileLogs");
  if (!aDir.Exists)
    aDir.Create();
}
```

11. Создайте каталог C:\TempWatch, а в нем — файл по имени temp.txt.

12. Запустите приложение. Если все пройдет успешно, щелкните на кнопке Browse и выберите файл C:\TempWatch\temp.txt.

13. Щелкните на кнопке Watch, чтобы начать отслеживание файла. Единственное изменение, которое вы увидите в приложении — это элемент управления типа метки, показывающий наблюдаемый файл.

14. Используя проводник Windows, перейдите в каталог C:\TempWatch. Откройте файл temp.txt в редакторе Notepad, добавьте некоторый текст к этому файлу и сохраните его.

15. Переименуйте файл.

16. Теперь можете проверить журнальный файл, чтобы увидеть в нем записи о проведенных изменениях. Перейдите к `C:\FileLogs\Log.txt` и откройте файл в редакторе Notepad. Вы должны увидеть описание изменений в выбранном для наблюдения файле (рис. 21.13).

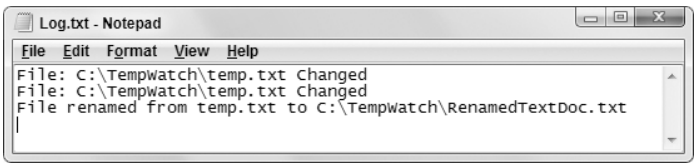


***Рис. 21.13**. Содержимое файла `Log.txt`*

Описание работы

Это несложное приложение демонстрирует работу класса `FileSystemWatcher`. Поэкспериментируйте со строкой, которая помещается в текстовое поле `txtLocation`. Если вы укажете для каталога `*.*`, будут отслеживаться все изменения в каталоге.

Большая часть кода приложения касается настройки объекта `FileSystemWatcher` для наблюдения за конкретным местоположением:

```
watcher.Path =Path.GetDirectoryName(txtLocation.Text);
watcher.Filter = Path.GetFileName(txtLocation.Text);
watcher.NotifyFilter = NotifyFilters.LastWrite |
   NotifyFilters.FileName | NotifyFilters.Size;
lblWatch.Text = "Watching " + txtLocation.Text;
// Начать наблюдение.
watcher.EnableRaisingEvents = true;
```

Сначала код устанавливает путь к каталогу для мониторинга. В нем используется новый объект, который пока еще не был показан — `System.IO.Path`. Это статический класс, во многом подобный статическому объекту `File`, который предоставляет множество статических методов для манипулирования и извлечения информации из строк, указывающих местоположение файлов. Сначала он применяется для извлечения имени каталога, которое пользователь ввел в текстовое поле, с помощью метода `GetDirectoryName()`.

Следующая строка кода устанавливает фильтр на объекте. Это может быть действительный файл — в этом случае будет отслеживаться только этот файл. Или же это может быть нечто вроде `*.txt`, и тогда будут отслеживаться все файлы `.txt` в указанном каталоге. Опять-таки, для извлечения информации из указанного местоположения файла применяется статический объект `Path`.

`NotifyFilter` — это комбинация значений перечисления `NotifyFilters`, которые указывают, из чего состоит изменение. В данном примере задано, что если изменяется временная метка последней операции записи, имя или размер файла, приложение получит уведомление об изменении. После обновления пользовательского интерфейса свойство `EnableRaisingEvents` устанавливается в `true` для запуска мониторинга.

Однако перед этим необходимо создать объект и установить обработчики событий:

```
this.watcher = new FileSystemWatcher();
  this.watcher.Deleted +=
new FileSystemEventHandler(this.OnDelete);
  this.watcher.Renamed +=
new RenamedEventHandler(this.OnRenamed);
  this.watcher.Changed +=
new FileSystemEventHandler(this.OnChanged);
  this.watcher.Created +=
new FileSystemEventHandler(this.OnCreate);
```

Подобным образом обработчики событий для наблюдаемого объекта привязываются к созданным приватным методам. Здесь настраиваются обработчики для событий, связанных с наблюдаемым объектом, когда файл удаляется, переименовывается, изменяется или создается. В своих методах вы сами решаете, как обрабатывать эти события. Обратите внимание, что уведомление получается *после* возникновения события.

В реальных методах обработки событий сообщение о событии просто записывается в журнальный файл. Очевидно, в зависимости от конкретного приложения, можно запрограммировать и более изощренную реакцию на событие. Когда в каталог добавляется файл, вы можете переместить его куда-то еще или прочитать содержимое и запустить новый процесс на основе прочитанной информации. Возможности практически безграничны!

Резюме

В этой главе вы узнали о потоках и о том, почему они используются в .NET Framework для доступа к файлам и другим последовательным устройствам. Вы познакомились с базовыми классами в пространстве имен `System.IO`, включая следующие:

- `File`
- `FileInfo`
- `FileStream`

Вы видели, что класс `File` предлагает множество методов для перемещения, копирования и удаления файлов, а `FileInfo` представляет физический файл на диске и предоставляет методы манипулирования этим файлом. Объект `FileStream` представляет файл, куда может быть выполнена запись, который может быть прочитан, либо то и другое. Вы также ознакомились с классами `StreamReader` и `StreamWriter` и увидели, насколько они полезны для записи потоков, а также научились читать и записывать в файлы произвольного доступа с применением класса `FileStream`. Основываясь на полученных знаниях, вы использовали классы из пространства имен `System.IO.Compression` для сжатия потоков при записи их на диск, а также изучили сериализацию объектов в файлы. И, наконец, вы построили целое приложение для слежения за файлами и каталогами, используя для этого класс `FileSystemWatcher`.

В этой главе были рассмотрены следующие вопросы.

- Как открывать файл, читать из него и записывать в файл.
- Разница между классами `StreamWriter` и `StreamReader`, с одной стороны, и классом `FileStream` — с другой.
- Как работать с файлами с разделителями для заполнения структур данных.
- Потоки со сжатием и распаковкой.
- Как сериализовать и десериализовать объекты.
- Мониторинг файловой системы классом `FileSystemWatcher`.

Упражнения

1. Какие пространства имен нужны приложению для работы с файлами?
2. Когда для записи файла следует использовать объект `FileStream` вместо объекта `StreamWriter`?
3. Какие методы класса `StreamWriter` позволяют читать данные из файлов, и что делает каждый их них?
4. Какой класс вы использовали бы для сжатия потока с применением алгоритма Deflate?

5. Как предотвратить сериализацию созданного вами класса?
6. Какие события представляет класс `FileSystemWatcher`, и для чего они служат?
7. Модифицируйте приложение `FileWatch`, построенное в этой главе, добавив возможность включения и отключения мониторинга файловой системы, не выходя из приложения.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Потоки	Поток — это абстрактное представление последовательного устройства, из которого можно читать или записывать по одному байту в единицу времени. Примером таких устройств могут служить файлы. Существуют два типа потоков — входные и выходные, которые предназначены, соответственно, для чтения и записи на устройства.
Классы доступа к файлам	В .NET Framework имеется множество классов, которые абстрагируют доступ к файловой системе, в том числе `File` и `Directory`, предназначенные для работы с файлами и каталогами через статические методы, а также `FileInfo` и `DirectoryInfo`, экземпляры которых можно создавать для представления специфических файлов и каталогов. Последняя пара классов полезна, когда выполняется множество операций над файлами и каталогами, поскольку эти классы не требуют передачи пути каждому вызову метода. Типичные операции, которые можно выполнять над файлами и каталогами, включают опрос и изменение свойств, создание, удаление и копирование.
Пути к файлам	Пути к файлам и каталогам могут быть абсолютными или относительными. Абсолютный путь дает полное описание расположения, начиная с корня устройства, которое содержит его; все родительские каталоги отделяются от дочерних с помощью обратной косой черты. Относительный путь похож, но начинается с определенной точки в файловой системе, такой как каталог, из которого запускается приложение (т.е. рабочий каталог). Для навигации по файловой системе часто используется `..` — псевдоним родительского каталога.
Объект `FileStream`	Объект `FileStream` предоставляет доступ к содержимому файла, как для чтения, так и для записи. Он обращается к данным файла на уровне байтов и потому не всегда является лучшим выбором для доступа к данным в файле. Экземпляр `FileStream` поддерживает байтовую позицию в файле, что позволяет осуществлять навигацию по содержимому файла. Возможность доступа к любой точке внутри файла подобным образом называется произвольным доступом.
Чтение и запись в потоки	Простейший способ чтения и записи данных в файл предусматривает использование классов `StreamReader` и `StreamWriter` в комбинации с `FileStream`. Вместо работы с байтами они позволяют считывать и записывать символьные и строковые данные. Эти типы предлагают хорошо известные методы для работы со строками, среди которых `ReadLine()` и `WriteLine()`. Имея дело со строками, упомянутые классы упрощают работу с файлами, содержащими данные, разделенные запятыми, которые являются общим способом представления структурированной информации.

Тема	Основные концепции
Сжатые файлы	Классы сжатых потоков `DeflateStream` и `GZipStream` можно использовать для чтения и записи сжатых данных в файлы. Эти классы работают с байтовыми данными в основном подобно `FileStream`, при этом, как и в случае с `FileStream`, к данным можно обращаться через классы `StreamReader` и `StreamWriter`, что упрощает код.
Сериализация объектов	Часто требуется сохранять и извлекать данные, которые представляют состояние объекта. Вместо написания собственного кода для сохранения и загрузки значений свойств можно применять технологию сериализации, которая обеспечивает сохранение и восстановление состояния объектов автоматически. Для этого понадобится помечать тип объекта как сериализируемый с помощью атрибута `Serializable`. Посредством других атрибутов можно также управлять тем, как члены будут сериализироваться, например, атрибут `NonSerialized` запрещает сериализацию конкретного члена.
Мониторинг файловой системы	Класс `FileSystemWatcher` можно использовать для мониторинга изменений в файловой системе. Имеется возможность отслеживать файлы и каталоги, а также при необходимости предоставлять фильтр для мониторинга модификации только файлов с указанным расширением. Экземпляры `FileSystemWatcher` уведомляют об изменениях за счет генерации событий, которые можно обрабатывать в собственном коде.

22

XML

В ЭТОЙ ГЛАВЕ...

- Как читать и писать Extensible Markup Language (XML)
- Правила, применимые к правильно оформленному XML
- Как проверять действительность ваших документов XML по двум типам схем: XSD и XDR
- Как использовать XML в ваших приложениях
- Как применять .NET для использования XML в ваших программах
- Как выполнять поиск в документах XML с помощью запросов XPath

Расширяемый язык разметки (Extensible Markup Language — XML) представляет собой технологию, которая привлекла к себе огромное внимание за последние несколько лет. Язык XML — не нов, и он определенно не был изобретен Microsoft для использования в среде .NET, но в Microsoft довольно рано оценили возможности, которые открывает этот язык для разработки. Как вы убедитесь, он охватывает широчайший круг применений — от описания конфигурации приложений до передачи информации между веб-службами.

XML — это способ хранения данных в простом текстовом формате, что означает, что он может быть прочитан почти на любом компьютере. Как уже было показано в предыдущих главах, посвященных программированию веб-приложений, это делает его непревзойденным форматом для передачи данных через Интернет. Его даже несложно читать и человеку!

Еще с самых первых версий Visual Studio .NET стало очевидным, что в Microsoft приложили немало усилий к разработке решений, использующих XML. Сегодня большинство приложений в .NET используют XML в той или иной форме — от файлов `.config` для хранения деталей конфигурации до файлов XAML, используемых в технологии Windows Presentation Foundations. Даже новые форматы документов, появившиеся в Office 2007, основаны на XML, хотя сами по себе приложения Office не являются приложениями .NET.

Детали XML могут оказаться очень сложными, поэтому мы не будем в них слишком углубляться. Однако базовый формат очень прост, и большинство задач даже не требуют глубоких знаний XML, поскольку Visual Studio обычно берет на себя большую часть работы, и вам редко придется писать XML-документы вручную. С учетом сказанного, XML чрезвычайно важен в мире .NET, поскольку применяется в качестве формата по умолчанию для передачи данных, а потому важно понимать хотя бы его основы.

Документы XML

Полный набор данных в XML известен как *документ XML*. Документом XML может быть физический файл на компьютере или просто строка в памяти. Однако он должен быть завершен сам по себе, и должен следовать определенным правилам (которые будут описаны ниже). Документ XML состоит из множества различных частей. Наиболее важные из них — элементы XML, которые содержат сами данные документа.

Элементы XML

Элемент XML состоит из открывающего дескриптора (имени элемента, заключенного в угловые скобки, например, `<myElement>`), данных внутри элемента и закрывающего дескриптора (такого же, как открывающий дескриптор, но с ведущим слэшем после открывающей скобки: `</myElement>`).

Например, определить элемент для хранения заголовка книги можно следующим образом:

```
<book>Tristram Shandy</book>
```

Если вы хотя бы немного знаете HTML, может показаться, что это очень похоже — и вы будете правы. Фактически HTML и XML разделяют почти один и тот же синтаксис. Главное отличие в том, что XML не имеет предопределенных элементов — вы сами выбираете имена для своих элементов, так что ничто не ограничивает их количество. Наиболее важный момент, о котором нужно помнить, состоит в том, что XML, несмотря на его название — это на самом деле вовсе не язык. Скорее, это стандарт для определения языков (известных как XML-приложения). Каждый язык имеет собственный отличный от других словарь — определенный набор элементов, которые могут применяться в документе, и структуру, которую могут принимать эти элементы. Как вы вскоре убедитесь, можно явно ограничивать допустимые элементы в документе XML. В качестве альтернативы можно

разрешить любые элементы и позволить самой программе, использующей документ, решать, какой должна быть структура.

Имена элементов чувствительны к регистру символов, поэтому <book> и <BOOK> трактуются как разные элементы. Это значит, что если вы попытаетесь закрыть элемент <book>, используя закрывающий дескриптор, записанный в другом регистре (например, </BOOK>), то XML-документ не будет правильным. Программы, читающие XML-документы и анализирующие их индивидуальные элементы, которые известны как XML-анализаторы (XML parsers), отклоняют любой документ, содержащий неправильный XML.

Элементы также могут содержать в себе другие элементы, поэтому несложно модифицировать элемент <book> так, чтобы включить информацию об авторе наряду в заголовком книги в двух подэлементах:

```
<book>
  <title>Tristram Shandy</title>
  <author>Lawrence Sterne</author>
</book>
```

Перекрытие элементов не допускается, поэтому подэлементы должны закрываться перед появлением закрывающего дескриптора родительского элемента. Это значит, например, что нельзя сделать так:

```
<book>
  <title>TristramShandy
    <author>LawrenceSterne
  </title></author>
</book>
```

Это запрещено, поскольку элемент <author> открыт внутри элемента <title>, но закрывающий дескриптор </title> находится перед закрывающим дескриптором </author>.

Существует одно исключение из правила, требующего, чтобы все элементы имели закрывающий дескриптор. Допускается иметь "пустые" элементы, не имеющие вложенных данных или текста. В этом случае можно просто добавлять закрывающий дескриптор немедленно после открывающего, как показано ниже:

```
<book></book>
```

Можно также применять сокращенный синтаксис, добавляя слэш закрывающего элемента в конец открывающего:

```
<book />
```

Атрибуты

Наряду с хранением данных в теле элемента можно также сохранять их внутри атрибутов, которые добавляются к открывающему элемент дескриптору. Атрибуты имеют следующую форму:

имя="*значение*"

Здесь значение атрибута *должно* быть заключено в одиночные или двойные кавычки. Например:

```
<book title="Tristram Shandy"></book>
```

или

```
<book title='Tristram Shandy'></book>
```

Оба приведенных варианта правильны, а следующий — нет:

```
<book title=Tristram Shandy></book>
```

Здесь может возникнуть вопрос: зачем нужны два способа хранения данных в XML? В чем разница между следующими двумя способами записи?

```
<book>
  <title>Tristram Shandy</title>
</book>
```

и

```
<book title="Tristram Shandy"></book>
```

Фактически между ними нет какого-то фундаментального отличия. Ни один, ни другой способ не дают большого преимущества друг перед другом. Элементы — более удачный выбор, если есть шанс, что понадобится добавлять информацию об этой части данных позднее — вы всегда можете добавить подэлемент или атрибут к элементу, но не можете делать то же самое с атрибутами. Возможно, элементы более читабельны и более элегантны (однако это — дело личного вкуса). В противоположность этому атрибуты потребляют меньше полосы пропускания, когда документ пересылается по сети без сжатия (при сжатии разница не велика), и удобны для хранения информации, которая не существенна каждому пользователю документа. Возможно, лучшим советом будет использовать и то, и другое, выбирая то, что наиболее удобно для вас, чтобы хранить определенный элемент данных, так что здесь не существует жестких правил.

Объявление XML

В дополнение к элементам и атрибутам XML-документы могут содержать множество других составных частей. Эти индивидуальные части документа XML известны под названием *узлов* (node); элементы, текст внутри элементов и атрибуты — все это узлы документа XML. Многие из них важны только в том случае, если вы действительно хотите углубиться в XML. Однако один тип узла появляется почти в каждом документе XML. Это *объявление XML*, и если вы его включаете, оно должно находиться в первом узле документа.

Объявление XML по своему формату подобно элементу, но имеет внутри себя вопросительные знаки и угловые скобки. Оно всегда имеет имя `xml` и всегда снабжено атрибутом по имени `version`; в настоящее время есть две возможные версии XML: 1.0 (первая редакция) и 1.1 (вторая редакция), но как ни странно, Visual Studio не поддерживает второй редакции. Следует отметить, что вторая редакция очень мало добавила к XML, что может понадобиться для нормального применения на платформе Windows, и консорциум World Wide Web Consortium (`www.w3c.org`) рекомендует по возможности пользоваться первой редакцией. Таким образом, простейшая из возможных форма объявления XML такова:

```
<?xml version="1.0"?>
```

Дополнительно объявление XML также включает атрибуты `encoding` (со значением, указывающим символьный набор, который должен быть использован для чтения документа, такой как `"UTF-16"`, чтобы показать, что документ использует 16-битный символьный набор Unicode) и `standalone` (со значениями `"yes"` или `"no"`, чтобы указать, зависит ли XML-документ от любых других файлов). Однако эти атрибуты не обязательны, и, скорее всего, в собственные XML-файлы вы будете включать только атрибут `version`.

Структура документа XML

Одна из наиболее важных характеристик XML состоит в том, что он предоставляет способ структурирования данных, который существенно отличается от реляционных баз данных. Большинство современных систем управления базами данных хранят информацию в таблицах, которые связаны друг с другом через значения определенных столбцов. Таблицы хранят данные в строках и столбцах: каждая строка представляет отдельную запись, а каждый столбец в ней — определенный элемент данных в этой строке. В отличие от

этого, данные XML структурированы иерархически, подобно организации папок и файлов в Windows Explorer. Каждый документ должен иметь единственный *корневой элемент*, внутри которого содержатся все элементы и текстовые данные. Если на вершине документа находится более одного элемента, то такой документ не считается правильным документом XML. Однако вы можете включить другие узлы XML в верхний уровень — особенно объявление XML. Таким образом, ниже показан правильный документ XML:

```
<?xml version="1.0"?>
<books>
  <book>Tristram Shandy</book>
  <book>Moby Dick</book>
  <book>Ulysses</book>
</books>
```

А этот фрагмент не является правильным документом XML:

```
<?xml version="1.0"?>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
```

Корневой элемент предоставляет значительную свободу относительно определения структуры данных. В отличие от реляционных данных, в которых каждая строка имеет одинаковое количество столбцов, здесь нет ограничений на число подэлементов, которые может содержать элемент. Вдобавок, хотя документы XML часто структурированы подобно реляционным данным, с элементом для каждой записи, документы XML вообще не нуждаются в каких-либо предопределенных структурах. Это — одно из основных отличий между традиционными реляционными базами данных и XML. В то время как реляционные базы данных всегда определяют структуру информации перед добавлением данных, информация может быть сохранена в XML без начальных накладных расходов, что очень удобно для хранения небольших блоков данных. Как вы вскоре убедитесь, вполне возможно представить структуру XML, но в отличие от реляционных баз данных, никто не требует указания этой структуры явным образом.

Пространства имен XML

Как известно из главы 9, любой может определять собственные классы C#, и любой может определять собственные элементы XML, что приводит к возникновению очевидной проблемы: как узнать, какие элементы к какому словарю относятся? Ответ — с помощью пространств имен. Точно так же, как вы определяете пространства имен для организации типов C#, вы используете пространства имен XML для определения собственных словарей XML. Это позволяет включать элементы из множества различных словарей в единственный документ XML, не рискуя неправильно интерпретировать их, потому что (например) два разных словаря определяют элемент `<customer>`.

Пространства имен XML могут быть достаточно сложными, поэтому в данном разделе мы не станем углубляться в детали. Тем не менее, базовый синтаксис прост. Определенные элементы или атрибуты ассоциированы с определенным пространством имен посредством префикса, за которым следует двоеточие. Например, `<wrox:book>` представляет элемент `<book>`, находящийся в пространстве имен `wrox`. Как вы узнаете, какое пространство имен представляет `wrox`? Чтобы этот подход работал, необходимо гарантировать уникальность каждого пространства имен. Простейший способ добиться этого — отобразить префиксы на нечто, уникальность чего известна, и именно так и делается. Где-то в документе XML потребуется ассоциировать любые префиксы пространств имен с *универсальным идентификатором ресурсов* (Uniform Resource Identifier — URI). URI существуют в нескольких ипостасях, но наиболее распространенный вид — простой веб-адрес вроде `www.wrox.com`.

Чтобы идентифицировать префикс с определенным пространством имен, используйте атрибут `xmlns:prefix` внутри элемента, установив это значение в уникальный URI, который идентифицирует пространство имен. Префикс может применяться в любом месте элемента, включая любые вложенные дочерние элементы:

```
<?xml version="1.0"?>
  <books>
    <book xmlns:wrox="http://www.wrox.com">
    <wrox:title>Beginning C#</wrox:title>
    <wrox:author>Karli Watson</wrox:author>
  </book>
</books>
```

Префикс `wrox:` можно использовать с элементами `<title>` и `<author>`, потому что они находятся внутри элемента `<book>`, где определен префикс. Однако если вы попытаетесь добавить этот префикс к элементу `<books>`, то XML станет неправильным, поскольку префикс для этого элемента не определен.

С использованием атрибута `xmlns` для элемента можно также определить пространство имен по умолчанию:

```
<?xml version="1.0"?>
<books>
  <book xmlns="http://www.wrox.com">
    <title>Beginning Visual C#</title>
    <author>Karli Watson </author>
    <html:img src="begvcsharp.gif"
              xmlns:html="http://www.w3.org/1999/xhtml" />
  </book>
</books>
```

Здесь пространство имен по умолчанию для элемента `<book>` определено как "`http://www.wrox.com`". Поэтому все, что находится внутри этого элемента, будет относиться к этому пространству имен, если только вы явно не укажете противоположное, добавив префикс другого пространства имен, как это делается для элемента `<img>` (когда вы устанавливаете пространство имен, используемое XML-совместимыми документами HTML).

Правильно оформленный и действительный XML

До сих пор речь шла о *правильном* XML. Фактически в XML делается различие между двумя формами правильности. Документы, соответствующие всем правилам стандарта XML, считаются *правильно оформленными*. Если документ XML не является правильно оформленным, то анализаторы не смогут интерпретировать его корректно, и отклонят такой документ. Чтобы быть правильно оформленным, документ должен отвечать перечисленным ниже требованиям.

- Иметь один, и только один, корневой элемент.
- Иметь закрывающие дескрипторы для каждого элемента (за исключением сокращенного синтаксиса, упомянутого ранее).
- Не иметь перекрывающихся элементов — все дочерние элементы должны полностью размещаться внутри родительского элемента.
- Иметь все атрибуты, заключенные в скобки.

Это не полный список требований, но он отражает наиболее распространенные ошибки, допускаемые программистами — новичками в XML. Однако документы XML могут соответствовать всем этим правилам и, тем не менее, не быть *действительными*. Вспомните, что ранее мы упоминали, что XML сам по себе не является языком, а скорее стандартом для определения приложений XML. Правильно оформленные документы XML просто со-

ответствуют стандарту XML; чтобы быть действительными, они также должны отвечать всем правилам, указанным для приложений XML. Не все анализаторы проверяют действительность документов; те, что делают это, называются *проверяющими анализаторами*. Чтобы проверить, соответствует ли документ правилам приложения, сначала необходим способ определения этих правил.

Проверка действительности документов XML

XML поддерживает два пути определения того, какие элементы и атрибуты могут быть помещены в документ, и в каком порядке: *определение типа документа* (Document Type Definitions — DTD) и *схема*.

DTD

Определения DTD используют отличный от XML синтаксис, унаследованный от предшественника XML, и постепенно заменяются схемами. DTD не позволяют указывать типы данных элементов и атрибуты, а потому являются не особо гибкими и не особенно широко используются в контексте .NET Framework. В противоположность этому, схемы применяются часто; они позволяют указывать типы данных, и написаны в XML-совместимом синтаксисе. К сожалению, схемы очень сложны, и существуют разные формы для их определения, даже в мире .NET!

Схемы

Существуют два формата схем, поддерживаемых .NET — язык определения схем XML (XML Schema Definition — XSD) и сокращенные схемы XML-данных (XML-Data Reduced — XDR).

Схемы XDR

Определение схем XDR — более старый стандарт, который принадлежит Microsoft; обычно он не используется и не распознается анализаторами, не принадлежащими Microsoft. XSD — открытый стандарт, рекомендованный W3C, и потому его определение присутствует здесь. Схемы могут как включаться внутрь XML-документа, так и храниться в отдельном файле. Вы должны быть хорошо знакомы с XML, прежде чем приступать к написанию схем, но стоит уметь распознавать главные элементы схемы, потому ниже будут изложены базовые принципы. Чтобы достичь понимания, мы рассмотрим простую схему XSD для простого документа XML, содержащую базовые детали о паре книг Wrox на тему C#. Этот документ XML можно найти в коде примеров для настоящей главы (`books.xml`):

```
<?xml version="1.0"?>
<books>
  <book>
    <title>Beginning Visual C# 2010</title>
    <author>Karli Watson</author>
    <code>7582</code>
  </book>
  <book>
    <title>Professional C# 2010</title>
    <author>Simon Robinson</author>
    <code>7043</code>
  </book>
</books>
```

Фрагмент кода **books.xml**

Схемы XSD

Элементы в схемах XSD должны относиться к пространству имен `http://www.w3.org/2001/XMLSchema`. Если это пространство имен не будет включено, элементы схемы не будут распознаны.

Чтобы ассоциировать документ XML со схемой XSD в другом файле, необходимо добавить элемент `schemalocation` к корневому элементу:

```
<?xml version="1.0"?>
<books schemalocation="file://C:\BegVCSharp\Chapter22\books.xsd">
  ...
</books>
```

Рассмотрим кратко пример схемы XSD:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="books">
    <complexType>
      <choice maxOccurs="unbounded">
        <element name="book">
          <complexType>
            <sequence>
              <element name="title" />
              <element name="author" />
              <element name="code" />
            </sequence>
          </complexType>
        </element>
      </choice>
      <attribute name="schemalocation" />
    </complexType>
  </element>
</schema>
```

Первое, что здесь бросается в глаза — это то, что в качестве пространства имен по умолчанию установлено пространство имен XSD. Это сообщает анализатору, что элементы документа относятся к схеме. Если вы не указываете это пространство имен, то анализатор будет считать, что это просто нормальные элементы XML, и не поймет, что он должен применять их для проверки.

Вся схема содержится внутри элемента под названием `<schema>` (с прописной буквы "s"; не забывайте, что регистр символов важен). Каждый элемент, который может появиться в документе, должен быть представлен элементом `<element>`. Этот элемент имеет атрибут `name`, указывающий имя элемента. Если элемент должен содержать в себе дочерние элементы, то для них должны быть предусмотрены дескрипторы `<element>` в элементе `<contextType>`. Внутри этого указывается, как именно дочерние элементы должны появляться.

Например, с помощью элемента `<choice>` указывается, что допустим любой выбор дочерних элементов, а посредством `<sequence>` — что дочерние элементы должны появляться в том же порядке, как они перечислены в схеме. Если элемент может встречаться более одного раза (подобно `<book>`), потребуется поместить атрибут `maxOccurs` внутрь родительского элемента. Установка его в `"unbounded"` означает, что элемент может появляться неограниченное количество раз. И, наконец, любые атрибуты должны быть представлены элементами `<attribute>`, включая атрибут `schemalocation`, который сообщает анализатору, где искать схему. Поместите это после конца списка дочерних элементов.

Теперь, ознакомившись с теоретическими основами XML, в следующем практическом занятии можно приступать к созданию документов XML. К счастью, среда Visual Studio выполняет массу черновой работы. Она даже создает схему XSD на основе документа XML, не требуя написания ни единой строки кода.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание документа XML в Visual Studio

Для создания документа XML выполните следующие шаги.

1. Откройте Visual Studio и выберите пункт меню File⇨New⇨File (Файл⇨Создать⇨Файл). Если вы не видите этого пункта, создайте новый проект, щелкните правой кнопкой мыши на проекте в Solution Explorer и выберите в контекстном меню добавление нового элемента.
2. В открывшемся диалоговом окне выберите элемент XML File (Файл XML) и щелкните на кнопке Add (Добавить). Среда Visual Studio создаст новый документ XML. Как видно на рис. 22.1, Visual Studio добавляет объявление XML, дополненное атрибутом `encoding` (среда также представляет атрибуты и элементы разными цветами).

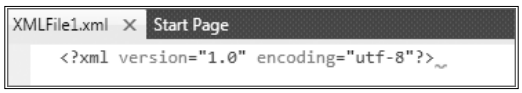


Рис. 22.1. *Автоматическое добавление объявления XML*

3. Сохраните файл, нажав <Ctrl+S> или выбрав пункт меню File⇨Save XMLFile1.xml (Файл⇨Сохранить `XMLFile1.xml`). Visual Studio запросит место для сохранения файла и его имя; сохраните его в папке `BegVCSharp\Chapter22` как `GhostStories.xml`.
4. Установите курсор в строке, находящейся под объявлением XML, и введите текст `<stories>`. Обратите внимание, что Visual Studio автоматически добавит закрывающий дескриптор, как только вы введете знак "больше" для закрытия открывающего дескриптора.
5. Введите следующий код в XML-файле и щелкните на кнопке Save (Сохранить):

```
<stories>
  <story>
    <title>A House in Aungier Street</title>
    <author>
      <name>Sheridan Le Fanu</name>
      <nationality>Irish</nationality>
    </author>
    <rating>eerie</rating>
  </story>
  <story>
    <title>The Signalman</title>
    <author>
      <name>Charles Dickens</name>
      <nationality>English</nationality>
    </author>
    <rating>atmospheric</rating>
  </story>
  <story>
    <title>The Turn of the Screw</title>
    <author>
      <name>Henry James</name>
      <nationality>American</nationality>
    </author>
    <rating>a bit dull</rating>
  </story>
</stories>
```

Фрагмент кода ***Chapter22\GhostStories.xml***

6. Теперь можно позволить Visual Studio создать схему, которая отвечает написанному XML-коду. Сделайте это, выбрав пункт меню XML⇨Create Schema (XML⇨Создать схему). Сохраните результирующий файл XSD, щелкнув на кнопке Save, как `GhostStories.XSD`.

7. Вернитесь к файлу XML и поместите следующий XML-код перед завершающим дескриптором `</stories>`:

```
<story>
  <title>Number 13 </title>
    <author>
      <name>M.R. James</name>
      <nationality>English</nationality>
    </author>
  <rating>mysterious</rating>
</story>
```

   Теперь вы получаете подсказки IntelliSense, когда начинаете набирать открывающие дескрипторы. Причина в том, что Visual Studio знает, как подключить вновь созданную схему XSD к набираемому файлу XML.

8. В Visual Studio можно установить связь между XML и одной или более схемами. Выберите пункт меню XML⇨Schemas (XML⇨Схемы). Это вызовет появление диалогового окна XML Schemas (Схемы XML), показанного на рис. 22.2. В начале длинного списка схем, которые распознает Visual Studio, вы увидите `GhostStories.XSD`. Слева от нее будет галочка зеленого цвета, указывающая, что схема используется текущим документом XML.

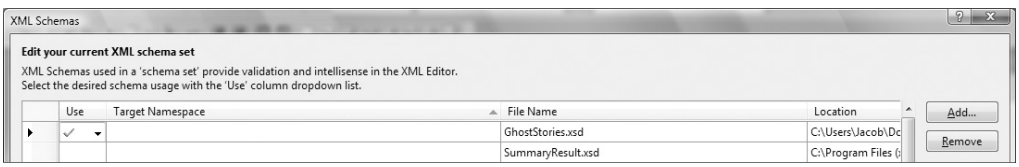


Рис. 22.2. *Диалоговое окно XML Schemas*

Диалоговое окно XML Schemas, показанное на рис. 22.2, содержит длинный список схем, распознаваемых VS, однако оно не запоминает автоматически использованные вами схемы. Если вы применяете схему многократно и не хотите искать ее всякий раз, когда она потребуется, можете скопировать ее в следующее место: `C:\Program Files\Microsoft Visual Studio 10.0\Xml\Schemas`. *Любая схема, скопированная в эту* папку, будет *отображаться в диалоговом окне XML Schemas.*

Использование XML в приложении

Теперь, когда вы узнали о том, как создаются документы XML, самое время применить полученные знания на практике. В состав .NET Framework включено множество пространств имен и классов, которые облегчают задачу чтения, манипулирования и записи XML. В последующих разделах будет описано множество классов, и мы посмотрим, как их применять для создания и программного манипулирования XML.

Объектная модель документа XML

Объектная модель документа XML (XML Document Object Model – XML DOM) – это набор классов, используемых для манипулирования XML интуитивно понятным образом. DOM – возможно, не самый быстрый способ чтения данных XML, но как только вы разберетесь в отношениях между классами и элементами документа XML, то обнаружите, что эта модель очень проста в применении.

Классы, составляющие DOM, находятся в пространстве имен `System.Xml`. В этом пространстве содержится несколько классов и пространств имен, но в данной главе мы сосредоточим внимание только на некоторых классах, которые позволяют легко манипулировать XML. Эти классы описаны в табл. 22.1.

Таблица 22.1. Важные классы из пространства имен `System.Xml`

Класс	Описание
`XmlNode`	Представляет отдельный узел в документе. Служит базовым для многих классов, описанных в этой главе. Если узел представляет корень документа XML, вы можете выполнить из него навигацию к любой позиции документа
`XmlDocument`	Расширяет класс `XmlNode`, но часто является первым объектом, который используется при работе с XML. Это связано с тем, что этот класс применяется для загрузки и сохранения данных с диска или еще откуда-либо
`XmlElement`	Представляет отдельный элемент в документе XML. Класс `XmlElement` унаследован от `XmlLinkedNode`, который, в свою очередь, унаследован от `XmlNode`
`XmlAttribute`	Представляет отдельный атрибут. Подобно классу `XmlDocument`, унаследован от `XmlNode`
`XmlText`	Представляет текст между открывающим и закрывающим дескрипторами
`XmlComment`	Представляет специальный тип узлов, которые не предназначены служить частью документа, а лишь предоставляют информацию читателю о частях документа
`XmlNodeList`	Представляет коллекцию узлов

Класс `XmlDocument`

Обычно первое, что должно делать ваше приложение с XML – это прочитать его из файла. Как описано в табл. 22.1, за это отвечает класс `XmlDocument`. Вы можете воспринимать `XmlDocument` как представление дискового файла, расположенное в памяти. Воспользовавшись классом `XmlDocument` для загрузки файла в память, вы можете получить корневой узел документа из него и приступить к чтению и манипуляциям кодом XML:

```
using System.Xml;
.
.
.
XmlDocument document = new XmlDocument();
document.Load(@"C:\BegVCSharp\Chapter22\books.xml");
```

Две строки кода создают новый экземпляр класса `XmlDocument` и загружают в него файл `books.xml`. Помните, что класс `XmlDocument` находится в пространстве имен `System.Xml`, и потому вы должны поместить строку `using System.Xml;` в начало кода.

В дополнение к загрузке и сохранению XML, класс `XmlDocument` также отвечает за поддержку самой структуры XML. Поэтому вы найдете в нем многочисленные методы, отве-

чающие за создание, изменение и удаление узлов дерева документа. Ниже мы рассмотрим некоторые из этих методов, но прежде чем представить эти методы правильно, вам следует узнать немного больше о другом классе — `XmlElement`.

Класс `XmlElement`

После загрузки документа в память с ним понадобится что-то делать. Свойство `DocumentElement` экземпляра `XmlDocument`, созданного в предыдущем коде, возвращает экземпляр `XmlElement`, представляющий корневой элемент `XmlDocument`. Этот элемент важен, поскольку обеспечивает доступ к каждому фрагменту информации в документе:

```
XmlDocument document = new XmlDocument();
document.Load(@"C:\BegVCSharp\Chapter22\books.xml");
XmlElement element = document.DocumentElement;
```

После получения корневого элемента документа вы готовы к использованию этой информации. Класс `XmlElement` содержит методы и свойства для манипуляции узлами и атрибутами дерева. Давайте сначала рассмотрим свойства, предназначенные для навигации по элементам XML, которые перечислены в табл. 22.2.

Таблица 22.2. Свойства, предназначенные для навигации по элементам XML

Свойство	Описание

`FirstChild`	Возвращает первый дочерний элемент после данного. Если вы вспомните файл `books.xml`, показанный ранее в этой главе, то в нем корневой элемент документа назывался `books`, а следующий за ним — `book`. Таким образом, в этом документе первым дочерним по отношению к корневому элементу `books` является `book`:
	`<books>  @@la Root node`
	`  <book>  @@la FirstChild`
	`nodeFirstChild` возвращает объект `XmlNode`, и вы должны проверить тип возвращенного узла, потому что маловероятно, что он всегда будет экземпляром `XmlElement`. В примере `books` дочерним по отношению к элементу `Title` фактически является узел `XmlText`, представляющий текст `Beginning Visual C#`.

`LastChild`	Работает точно так же, как свойство `FirstChild`, за исключением того, что возвращает последний дочерний элемент текущего узла. В случае примера `books` последним дочерним элементом узла `books` также будет узел `book`, но представляющий книгу `Professional C# 2010`.
	`<books> @@la Root node`
	`  <book> @@la FirstChild`
	`  <title>Beginning Visual C# 2010</title>`
	`  <author>Karli Watson</author>`
	`  <code>7582</code>`
	`  </book>`
	`  <book> @@la LastChild`
	`  <title>Professional C# 2010</title>`
	`  <author>Simon Robinson</author>`
	`  <code>7043</code>`
	`  </book>`
	`</books>`

`ParentNode`	Возвращает родителя текущего узла. В примере `books` узел `books` является родителем обоих узлов `book`

Окончание табл. 22.2

Свойство	Описание
`NextSibling`	В то время как свойства `FirstChild` и `LastChild` возвращают листовой узел текущего узла, узел `NextSibling` возвращает следующий узел, имеющий того же родителя. В случае примера `books` это означает, что `NextSibling` элемента `title` вернет элемент `author`, а вызов `NextSibling` на нем вернет элемент `code`
`HasChildNodes`	Позволяет проверить наличие дочерних элементов у текущего элемента, не получая значения от `FirstChild` и не проверяя его на равенство `null`

Используя четыре свойства из табл. 22.2, можно проходить по всему документу `XmlDocument`, как показано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Цикл прохода по всем узлам документа XML

В этом примере создается небольшое приложение Windows Forms, которое пройдет циклом по всем узлам документа XML и выведет имена элементов, а в случае элемента `XmlText` — текст, содержащийся в элементе. В этом коде используется файл `books.xml`, который был представлен в разделе "Схемы" ранее в главе; если вы не создали этот файл во время изучения упомянутого раздела, найдите его в коде примеров для данной главы.

1. Начните с создания нового приложения Windows Forms, выбрав пункт меню File⇨New⇨Project (Файл⇨Создать⇨Проект). В открывшемся диалоговом окне выберите элемент Windows Application (Приложение Windows). Назовите проект `LoopThroughXmlDocument`.
2. Спроектируйте форму, как показано на рис. 22.3, перетащив на нее элементы управления `ListBox` и `Button`.

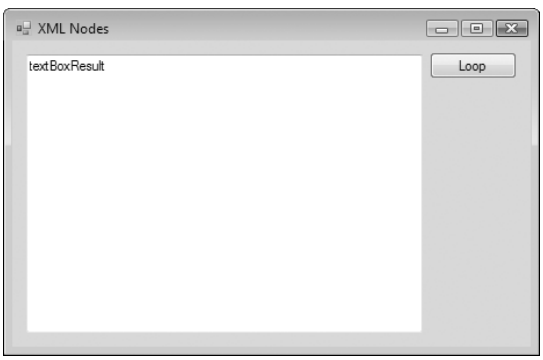


Рис. 22.3. *Форма приложения* `LoopThroughXmlDocument`

3. Назначьте элементу `ListBox` имя `listBoxXmlNodes`, а кнопке — `buttonLoopThroughDocument`. Установите свойство `Multiline` элемента `ListBox` в `true`, а свойство `Scrollbars` — в `Vertical`.
4. Дважды щелкните на кнопке и введите приведенный ниже код. Не забудьте добавить строку `using System.Xml;` в начало файла.

```
private void buttonLoopThroughDocument_Click(object sender, EventArgs e)
{
  XmlDocument document = new XmlDocument();
  document.Load(@"C:\BegVCSharp\Chapter22\Books.xml");
  textBoxResult.Text = FormatText(document.DocumentElement as XmlNode, "", "");
}

private string FormatText(XmlNode node, string text, string indent)
{
  if (node is XmlText)
  {
    text += node.Value;
    return text;
  }
  if (string.IsNullOrEmpty(indent))
    indent = "";
  else
  {
    text += "\r\n" + indent;
  }
  if (node is XmlComment)
  {
     text += node.OuterXml;
     return text;
  }
  text += "<" + node.Name;
  if (node.Attributes.Count > 0)
  {
     AddAttributes(node, ref text);
  }
  if (node.HasChildNodes)
  {
     text += ">";
     foreach (XmlNode child in node.ChildNodes)
     {
        text = FormatText(child, text, indent + " ");
     }
     if (node.ChildNodes.Count == 1 &&
         (node.FirstChild is XmlText || node.FirstChild is XmlComment))
        text += "</" + node.Name + ">";
     else
        text += "\r\n" + indent + "</" + node.Name + ">";
  }
  else
     text += " />";
  return text;
}

private void AddAttributes(XmlNode node, ref string text)
{
  foreach (XmlAttribute xa in node.Attributes)
  {
     text += " " + xa.Name + "='" + xa.Value + "'";
  }
}
```

Фрагмент кода ***Chapter22\LoopThroughXmlDocument\Form1.cs***

5. Запустите приложение и щелкните на кнопке Loop (Цикл). Вы должны получить результат, показанный на рис. 22.4.

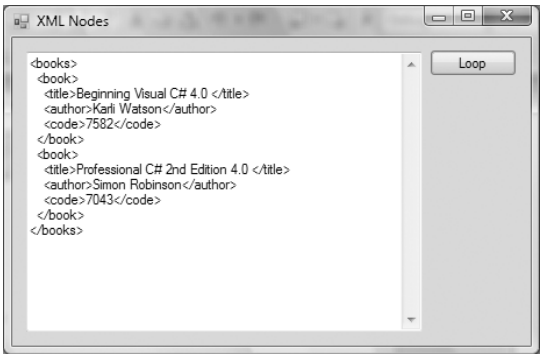


Рис. 22.4. *Результат работы приложения* `LoopThroughXmlDocument`

Описание работы

После щелчка на кнопке первым делом вызывается метод `Load` класса `XmlDocument`. Этот метод загружает XML из файла в экземпляр `XmlDocument`, который впоследствии используется для доступа к элементам XML. Затем вызывается метод, позволяющий рекурсивно пройти циклом по XML, которому передается корневой узел документа XML. Корневой элемент получается из свойства `DocumentElement` класса `XmlDocument`. Помимо проверки на `null` переданного в метод `RecurseXmlDocument` параметра `root`, первая строка, на которую следует обратить внимание — это оператор `if`:

```
if (node is XmlText)
{
  ...
}
```

Вспомните, что операция `is` позволяет проверить тип объекта и возвращает `true`, если экземпляр относится к указанному типу. Несмотря на то что узел `root` объявлен как `XmlNode`, это всего лишь базовый тип объектов, с которыми вы будете работать. Используя операцию `is` для проверки типа объектов, код может определить тип объекта во время выполнения и вести себя соответственно.

Внутри метода `FormatText` генерируется текст для текстового поля. Вы должны знать тип текущего экземпляра `root`, потому что информация, которую требуется отобразить, получается по-разному для разных элементов: необходимо отобразить имена `XmlElement` и значения `XmlText`.

Изменение значений узлов

Прежде чем посмотреть, как изменяется значение узла, важно понять, что значение узла очень редко бывает простым. Фактически вы столкнетесь с тем, что почти все классы, унаследованные от `XmlNode`, включают свойство по имени `Value`, которое редко возвращает что-то полезное. Хотя поначалу это несколько разочаровывает, впоследствии вы сочтете это достаточно логичным. Рассмотрим ранее приведенный пример `books`:

```
<books>
  <book>
    <title>Beginning Visual C# 2010</title>
    <author>Karli Watson</author>
    <code>7582</code>
  </book>
  <book>
</books>
```

Каждая отдельная пара дескрипторов представлена в DOM как узел. Вспомните, что когда вы проходили циклом по всем узлам документа, то встречали множество узлов `XmlElement` и три элемента `XmlText`. Узлами `XmlElement` в этом XML являются `<books>`, `<book>`, `<title>`, `<author>` и `<code>`. Узлы `XmlText` представляют собой текст, находящийся между начальным и конечным дескрипторами `title`, `author` и `code`. Хотя можно утверждать, что значениями узлов `title`, `author` и `code` является текст между дескрипторами, этот текст сам по себе является узлом, и как таковой, содержит в себе значение. Другие узлы, очевидно, не имеют никаких ассоциированных с ними значений помимо вложенных узлов.

Следующая строка кода находится в блоке `if`, расположенном ближе к началу кода в показанном ранее методе `FormatText`. Она выполняется, когда текущим узлом является `XmlText`.

```
text += node.Value;
```

Здесь можно видеть, что свойство `Value` узла `XmlText` используется для получения значения узла.

Узлы типа `XmlElement` возвращают `null`, если вы используете свойство `Value`, но можно получить информацию, находящуюся между начальным и конечным дескрипторами `XmlElement`, если применить один из двух других методов: `InnerText` и `InnerXml`. Это значит, что можно манипулировать значением узлов, используя два метода и свойство, которые описаны в табл. 22.3.

Таблица 22.3. Средства для манипулирования значениями узлов

Средство	Описание
`InnerText`	Получает текст всех дочерних по отношению к текущему узлов и возвращает его в виде единственной соединенной строки. Это значит, что если вы получаете значение `InnerText` из узла `book` предыдущего документа XML, будет возвращена строка `Beginning Visual C# 2010Karli Watson7582`. Если вы получите `InnerText` узла `title`, будет возвращено только `Beginning Visual C# 2010`. С помощью этого метода можно устанавливать текст, но будьте осторожны, делая это, поскольку если вы установите текст неверного узла, то можете переписать информацию, которую не нужно было бы изменять
`InnerXml`	Возвращает текст, как и `InnerText`, но также возвращает все дескрипторы. Таким образом, если вы получите значение `InnerXml` узла `book`, то результатом будет следующая строка: `<title>Beginning Visual C# 2010 </title><author>Karli Watson</author><code>7582</code>`. Как видите, это может быть довольно удобно, если есть строка, содержащая XML, которую необходимо включить непосредственно в документ XML. Однако в этом случае вы полностью отвечаете за содержимое этой строки, и если вставите неправильный XML, то приложение сгенерирует исключение
`Value`	Обеспечивает самый “чистый” способ манипуляции информацией в документе, но, как упоминалось ранее, только несколько классов в действительности возвращают нечто полезное при получении значения. Классы, возвращающие нужный текст, таковы: • `XmlText` • `XmlComment` • `XmlAttribute`

Вставка новых узлов

Теперь, когда вы увидели, как перемещаться по документу XML и даже получать значения его элементов, давайте рассмотрим, как изменить структуру документа, добавляя узлы к документу `books.xml`, с которым мы имели дело до сих пор.

Чтобы вставлять новые элементы в список, нужно ознакомиться с новыми методами классов `XmlDocument` и `XmlNode`, которые перечислены в табл. 22.4. В классе `XmlDocument` определены методы, позволяющие создавать новые экземпляры `XmlNode` и `XmlElement`, что замечательно, поскольку оба эти класса имеют только защищенные (`protected`) конструкторы, а это значит, что вы не можете создать экземпляр каждого из них непосредственно операцией `new`.

Методы, перечисленные в табл. 22.5, используются для создания узлов, но после вызова любого из них вы должны что-то делать с ними, прежде чем они станут интересными. Немедленно после создания узлы не содержат дополнительной информации, и они еще не вставлены в документ. Чтобы сделать то или другое, вы должны использовать методы, находящиеся в любом классе-наследнике `XmlNode` (включая `XmlDocument` и `XmlElement`), которые описаны в табл. 22.4.

Таблица 22.4. Методы классов, унаследованных от `XmlNode`

Метод	Описание
`AppendChild`	Добавляет дочерний узел к узлу `XmlNode` или производного от него типа. Помните, что добавляемый узел появляется в конце списка дочерних того узла, на котором метод был вызван. Если вас не заботит порядок дочерних элементов, то нет проблем; если же порядок важен, добавляйте узлы в нужном порядке
`InsertAfter`	Указывает точное место вставки нового узла. Метод принимает два параметра: первый — новый узел, а второй — узел, после которого новый узел должен быть вставлен
`InsertBefore`	Работает подобно `InsertAfter`, за исключением того, что новый узел вставляется перед узлом, который вы передаете в виде ссылки

Таблица 22.5. Методы для создания узлов

Метод	Описание
`CreateNode`	Создает узел любого вида. Существуют три перегрузки этого метода: две из них позволяют создавать узлы типа из перечисления `XmlNodeType`, а одна дает возможность указывать тип используемого узла в виде строки. Если только вы не абсолютно уверены в том, что не укажете ошибочно в виде строки тип узла, отсутствующий в перечислении, настоятельно рекомендуется отдавать предпочтение перегрузкам, принимающим аргумент типа перечисления. Метод возвращает экземпляр `XmlNode`, который может быть явно приведен к соответствующему типу
`CreateElement`	Версия `CreateNode`, создающая только узлы типа `XmlDocument`
`CreateAttribute`	Версия `CreateNode`, создающая только узлы типа `XmlAttribute`
`CreateTextNode`	Создает узлы типа `XmlTextNode`
`CreateComment`	Этот метод включен здесь для того, чтобы описать типы узлов, которые могут быть созданы. Этот метод не создает узел, являющийся частью данных, представленных документом XML, а создает комментарий, предназначенный для чтения человеком. Можно также извлекать комментарии при чтении документа приложением

В следующем практическом занятии представлен пример, построенный на базе предыдущего, который вставляет узел `book` в документ `books.xml`. В примере нет кода для очистки документа (пока), поэтому, запустив его несколько раз, вы получите несколько идентичных узлов.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание узлов

Выполните следующие шаги, чтобы добавить узел в документ `books.xml`.

1. Добавьте кнопку, разместив ее ниже существующей кнопки в форме, и назовите ее `buttonCreateNode`.
2. Дважды щелкните на новой кнопке и введите следующий код:

```
private void buttonCreateNode_Click(object sender, EventArgs e)
{
  // Загрузка документа XML.
  XmlDocument document = new XmlDocument();
  document.Load(@"C:\BegVCSharp\Chapter22\Books.xml");

  // Получение корневого элемента.
  XmlElement root = document.DocumentElement;

  // Создание новых узлов.
  XmlElement newBook = document.CreateElement("book");
  XmlElement newTitle = document.CreateElement("title");
  XmlElement newAuthor = document.CreateElement("author");
  XmlElement newCode = document.CreateElement("code");
  XmlText title = document.CreateTextNode("Beginning Visual C# 2008");
  XmlText author = document.CreateTextNode("Karli Watson et al");
  XmlText code = document.CreateTextNode("1234567890");
  XmlComment comment = document.CreateComment("The previous edition");

  // Вставка новых элементов.
  newBook.AppendChild(comment);
  newBook.AppendChild(newTitle);
  newBook.AppendChild(newAuthor);
  newBook.AppendChild(newCode);
  newTitle.AppendChild(title);
  newAuthor.AppendChild(author);
  newCode.AppendChild(code);
  root.InsertAfter(newBook, root.FirstChild);
  document.Save(@"C:\BegVCSharp\Chapter22\Books.xml");
}
```

Фрагмент кода ***Chapter22\InsertingNodes\Form1.cs***

3. Запустите приложение и щелкните на кнопке Create Node (Создать узел). Затем щелкните на кнопке Loop (Цикл). Должно получиться диалоговое окно, показанное на рис. 22.5.

Существует один важный тип узла, который не был создан в предыдущем примере, а именно — `XmlAttribute`. Оставим это в качестве упражнения для данной главы.

Описание работы

Код в методе `buttonCreateNode_Click` — это место, где происходит собственно создание узлов. Он создает восемь новых узлов, четыре из которых имеют тип `XmlElement`, три — тип `XmlText` и один — тип `XmlComment`.

Все узлы создаются методом, инкапсулирующим экземпляр `XmlDocument`. Узлы `XmlElement` создаются с помощью метода `CreateElement`, узлы `XmlText` — посредством `CreateTextNode`, а узел `XmlComment` — методом `CreateComment`.

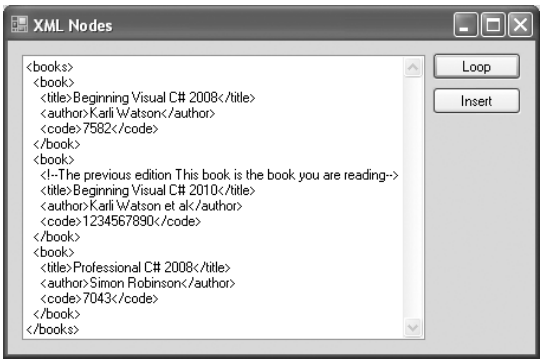


Рис. 22.5. *Добавление нового узла*

После создания эти узлы еще должны быть вставлены в дерево XML. Это делается вызовом метода `AppendChild` на элементе, для которого новый узел должен стать дочерним. Единственное исключение — узел `book`, который является корневым узлом для всех новых узлов. Этот узел вставляется в дерево с помощью метода `InsertAfter` корневого объекта. В то время как все узлы, вставленные методом `AppendChild`, всегда становятся последними в списке дочерних узлов, `InsertAfter` позволяет должным образом позиционировать узел.

Удаление узлов

Теперь, когда известно, как создаются новые узлы, все, что остается изучить — это как их удалить. Все классы, унаследованные от `XmlNode`, имеют методы, которые позволяют удалять узлы из документа (табл. 22.6).

Таблица 22.6. Методы удаления узлов

Метод	Описание
`RemoveAll`	Удаляет все дочерние узлы того узла, на котором метод вызван. Чуть менее очевидно то, что он также удаляет все атрибуты узла, поскольку они также считаются дочерними узлами
`RemoveChild`	Удаляет единственный дочерний узел того узла, на котором вызван. Метод возвращает узел, который был удален из документа, так что его можно при необходимости вставить заново

Следующее короткое практическое занятие расширяет приложение Windows Forms, которое создавалось на протяжении двух последних примеров, добавляя возможность удаления узлов. Пока оно только ищет последний экземпляр узла `book` и удаляет его.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Удаление узлов

Следующие шаги позволяют найти и удалить последний экземпляр узла `book`.

1. Добавьте новую кнопку, разместив ее под двумя существующими, и назовите ее `buttonDeleteNode`. Установите ее свойство `Text` в `Delete Node`.
2. Дважды щелкните на новой кнопке и введите следующий код:

```
private void buttonDeleteNode_Click(object sender, EventArgs e)
{
  // Загрузить документ XML.
```

```
    XmlDocument document = new XmlDocument();
    document.Load(@"C:\BegVCSharp\Chapter22\Books.xml");

    // Получить корневой элемент.
    XmlElement root = document.DocumentElement;

    // Найти узел. Корнем является дескриптор <books>, поэтому
    // его последний дочерний узел и будет последним узлом <book>.
    if (root.HasChildNodes)
    {
      XmlNode book = root.LastChild;

      // Удалить дочерний узел.
      root.RemoveChild(book);

      // Сохранить документ на диск.
      document.Save(@"C:\BegVCSharp\Chapter22\Books.xml");
    }
  }
```

3. Запустите приложение. После щелчка на кнопке Delete Node (Удалить узел), а затем на кнопке Loop (Цикл) последний узел в дереве исчезнет.

Описание работы

После начальных шагов по загрузке XML в объект `XmlDocument` вы проверяете корневой элемент, чтобы увидеть, есть ли какие-то дочерние элементы в загруженном XML. Если они есть, с помощью свойства `LastChild` класса `XmlElement` извлекается последний дочерний узел. После этого удаление элемента осуществляется простым вызовом `RemoveChild` с передачей ему экземпляра элемента, который требуется удалить. В данном случае — это последний дочерний узел корневого элемента.

Выбор узлов

Теперь вы знаете, как передвигаться вперед и назад по документу XML, как манипулировать значениями документа, как создавать новые узлы и как затем удалять их. Остается в этой главе изложить только одно: как выбирать узлы напрямую, не выполняя обход всего дерева.

Класс `XmlNode` включает два метода, часто используемых для выбора узлов из документа, не обходя их все. Это `SelectSingleNode` и `SelectNodes` (табл. 22.7) — оба они применяют для выборки узлов специальный язык запросов, называемый XPath. Ниже вы узнаете о нем.

Таблица 22.7. Методы выбора узлов

Метод	Описание
`SelectSingleNode`	Выбирает единственный узел. Если вы создаете запрос, выбирающий более одного узла, возвращается только первый из них
`SelectNodes`	Возвращает коллекцию узлов в форме класса `XmlNodeList`

XPath

XPath — это язык запросов для документов XML, подобно тому, как SQL — язык запросов для реляционных баз данных. Он используется двумя методами, описанными в табл. 22.7, для того, чтобы позволить избежать необходимости обхода всего дерева документа XML. Однако о нем следует поговорить особо, поскольку его синтаксис не имеет ничего общего с SQL или C#.

XPath — довольно обширный язык, и мы опишем здесь лишь малую его часть, достаточную для того, чтобы можно было приступить к извлечению узлов. Если вы хотите узнать больше, загляните по адресу `www.w3.org/TR/xpath` *и на страницы справочной системы Visual Studio.*

Чтобы увидеть XPath в действии, мы воспользуемся несколько расширенной версией файла `books.xml`. Его содержимое будет приведено в практическом занятии раздела "Выбор узлов" далее в этой главе (он также доступен в виде файла `Elements.xml` в коде примеров для этой главы).

В табл. 22.8 перечислены некоторые из наиболее распространенных операций, которые можно выполнять с XPath. Если не установлено иное, пример запроса XPath выполняет выборку относительно узла, к которому он применен. Когда необходимо иметь имя узла, можете предполагать, что текущим узлом является `<book>` из документа XML.

Таблица 22.8. Наиболее распространенные операции, выполняемые с помощью XPath

Назначение	Пример запроса XPath
Выбрать текущий узел	`.`
Выбрать родительский узел для текущего узла	`..`
Выбрать все дочерние узлы текущего узла	`*`
Выбрать все дочерние узлы с указанным именем, в данном случае — `title`	`title`
Выбрать атрибут текущего узла	`@Type`
Выбрать все атрибуты текущего узла	`@*`
Выбрать дочерний узел по индексу, в данном случае — второй узел `element`	`element[2]`
Выбрать все текстовые узлы текущего узла	`text{}`
Выбрать одного или более внуков текущего узла	`element/text()`
Выбрать все узлы в документе с определенным именем, в данном случае — все узлы `mass`	`//mass`
Выбрать все узлы в документе с определенным именем и определенным именем родителя. В данном случае — узлы `element` с родителем `name`	`//element/name`
Выбрать все узлы в документе с определенным именем и определенным именем родителя, в данном случае — элемент с именем `Hydrogen`	`//element[name='Hydrogen']`
Выбрать узел с определенным значением атрибута `Type`, в данном случае `Type` равен `Noble Gas`	`//element[@Type='Noble Gas']`

В следующем практическом занятии будет создано небольшое приложение, которое позволит выполнить и увидеть результаты предопределенных запросов, а также вводить собственные запросы.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Выбор узлов

Как упоминалось ранее, в этом примере используется новый файл XML по имени `Elements.xml`, входящий в состав кода примеров. Разумеется, код можно ввести и самостоятельно:

```
<?xml version="1.0"?>
<elements>
  <!--Первый элемент Non-Metal-->
  <element Type="Non-Metal">
    <name>Hydrogen</name>
    <symbol>H</symbol>
    <number>1</number>
    <specification>
      <mass>1.007825</mass>
      <density>0.0899 g/cm3</density>
    </specification>
  </element>
  <!--Первый элемент Noble Gas-->
  <element Type="Noble Gas">
    <name>Helium</name>
    <symbol>He</symbol>
    <number>2</number>
    <specification>
      <mass>4.002602</mass>
      <density>0.1785 g/cm3</density>
    </specification>
  </element>
  <!--Первый элемент Halogen-->
  <element Type="Halogen">
    <name>Fluorine</name>
    <symbol>F</symbol>
    <number>9</number>
    <specification>
      <mass>18.998404</mass>
      <density>1.696 g/cm3</density>
    </specification>
  </element>
  <element Type="Noble Gas">
    <name>Neon</name>
    <symbol>Ne</symbol>
    <number>10</number>
    <specification>
      <mass>20.1797</mass>
      <density>0.901 g/cm3</density>
    </specification>
  </element>
</elements>
```

Фрагмент кода ***Chapter22\XpathQuery\Elements.xml***

Сохраните этот XML-файл под именем `Elements.xml`. Не забудьте изменить путь к файлу в следующем коде.

Выполните перечисленные ниже шаги для создания приложения Windows Forms с возможностью выполнения запросов.

1. Создайте новое приложение Windows Forms и назовите его `XPathQuery`.
2. Постройте диалоговое окно, показанное на рис. 22.6. Назовите все элементы управления, как показано на рисунке, за исключением кнопки, которая должна иметь имя `buttonExample`, и установите свойство `Scrollbars` элемента `textBoxResult` в `Vertical`.

Рис. 22.6. *Диалоговое окно приложения* `XPathQuery`

3. Щелкните правой кнопкой мыши на форме и выберите в контекстном меню пункт View Code (Просмотреть код). Добавьте следующую директиву `using`:

```
using System.Xml;
```

4. Добавьте два приватных поля для хранения документа и текущего узла и инициализируйте их в конструкторе:

```
private XmlDocument mDocument;
public Form1()
{
  InitializeComponent();

  document = new XmlDocument();
  document.Load(@"C:\BegVCSharp\Chapter22\Elements Subset.xml");
}
```

5. Вам понадобятся еще несколько вспомогательных методов для отображения результата запросов в текстовом окне `textBoxResult`:

```
private void Update(XmlNodeList nodes)
{
  if (nodes == null || nodes.Count == 0)
  {
    textBoxResult.Text = "The query yielded no results"; // запрос не дал результатов
    return;
  }
  string text = "";
  foreach (XmlNode node in nodes)
  {
    text = FormatText(node, text, "") + "\r\n";
  }
  textBoxResult.Text = text;
}
```

6. Модифицируйте конструктор для отображения всего содержимого файла XML при запуске приложения:

```
public Form1()
{
  InitializeComponent();
  document = new XmlDocument();
  document.Load(@"C:\BegVCSharp\Chapter22\Elements Subset.xml");
  Update(document.DocumentElement.SelectNodes("."));
}
```

7. Скопируйте и вставьте два метода `FormatText` и `AddAttributes` из предыдущего практического занятия в новый проект.

8. И, наконец, вставьте код, который выполнится, когда пользователь введет что-нибудь в текстовом поле:

```
private void buttonExecute_Click(object sender, EventArgs e)
{
  try
  {
    XmlNodeList nodeList = mCurrentNode.SelectNodes(textBoxQuery.Text);
    Update(nodes);
  }
  catch (Exception err)
  {
    textBoxResult.Text = err.Message;
  }
}
```

9. Запустите приложение и введите следующий запрос в текстовом поле `textBoxQuery` для выбора узла элемента, содержащего узел с текстом "Hydrogen":

```
element[name='Hydrogen'];
```

Описание работы

Метод `buttonExecute_Click` — это метод, выполняющий запросы. Поскольку заранее не известно, породит ли запрос, введенный в `textBoxQuery`, один или множество узлов, необходимо использовать метод `SelectNodes`. Это либо вернет объект `XmlNodeList`, либо сгенерирует одно из исключений, связанных с `XPath`, если используемый запрос окажется неверным.

Метод `Update` отвечает за циклический проход по содержимому `XmlNodeList`, выбранному с помощью `SelectNodes`. Он вызывает `FormatText` из предыдущих примеров с каждым из узлов, а `FormatText` отвечает за рекурсивный обход дерева узлов и создание читабельного текста, который можно отобразить в элементе управления `textBoxResult`.

В упражнениях, приведенных в конце главы, вы найдете множество дополнительных запросов XPath для опробования. Прежде чем вы введете их в приложение `XPathQuery`, чтобы увидеть результат, попробуйте представить, каким он будет.

Резюме

В этой главе вы ознакомились с расширяемым языком разметки (XML) — текстовым форматом для хранения и извлечения данных. Вы изучили правила, которые нужно соблюдать для того, чтобы гарантировать создание правильно оформленных документов XML, а также узнали, как проверить их на соответствие схемам XSD и XDR.

После изучения основ XML вы увидели, как XML может применяться в коде C# с использованием Visual Studio. И, наконец, вы узнали о применении XPath для формулирования запросов к XML.

В следующей главе вы узнаете об очень интересном языке запросов — LINQ. Этот язык может также применяться для опроса XML, но это выходит за рамки настоящей книги. Прежде чем двигаться дальше, постарайтесь выполнить следующие упражнения.

Упражнения

1. Измените пример из раздела "Создание узлов" так, чтобы в узел `book` вставлялся атрибут по имени `Pages` со значением `1000`.
2. Представьте вывод следующих запросов XPath, а затем сравните ваши результаты, вводя запросы в приложение `XPathQuery` из раздела "Выбор узлов". Помните, что все запросы выполняются на `DocumentElement`, который представляет собой узел элемента.

```
//elements
element
element[@Type='Noble Gas']
//mass
//mass/..
element/specification[mass='20.1797']
element/name[text()='Neon']
```

3. На многих системах Windows средством просмотра XML по умолчанию является веб-браузер. Если вы используете Internet Explorer, то, загрузив в него файл `Elements.xml`, вы увидите симпатично сформатированное представление XML. Не правда ли, было бы идеально отображать XML, получаемый из запросов XPath, в браузерном элементе управления вместо текстового поля?

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Синтаксис XML	Документы XML создаются из объявления XML, пространств имен XML, а также элементов и атрибутов XML. Объявление XML определяет версию XML. Пространства имен XML служат для определения словарей и используемых элементов и атрибутов XML, которые вместе образуют содержимое документа XML.
Правильно оформленный XML	Правильно оформленный XML — это XML, который удовлетворяет базовым синтаксическим правилам XML. Документ называется правильно оформленным, когда в нем имеется строго один корневой элемент, каждый элемент имеет закрывающий дескриптор, отсутствуют перекрытия элементов (все дочерние элементы должны быть полностью вложенными в родительские) и все атрибуты заключены в кавычки. Все средства считывания XML могут читать правильно оформленный XML, но очень немногие позволяют читать документы, не являющиеся правильно оформленными. Строго говоря, если документ содержит неправильно оформленные дескрипторы, он не является документом XML.
Действительный XML	Действительный XML — это XML, который является правильно оформленным и может быть проверен на предмет возможности его генерации из схемы. Действительность XML важна, поскольку она позволяет делать предположения относительно содержимого документа XML. Это дает возможность работать с документами, сгенерированными другими, имея уверенность, что структура и имена внутри документа являются теми, которые ожидались.
Схема XML	Схема XML используется для определения структуры документов XML. Схемы особенно полезны, когда нужно обмениваться информацией с третьими сторонами. Согласившись со схемой для обмениваемых данных, стороны получают возможность проверять, являются ли документы действительными.
Использование XML в программах	В настоящее время XML широко используется в .NET, и платформа .NET Framework предоставляет набор классов для создания и манипулирования кодом XML. Документы XML можно применять для хранения конфигурации приложений, записи данных на диск, пересылки информации по сети и т.п.
XPath	Язык XPath является одним из возможных способов запроса данных из документов XML. Для использования XPath потребуется знать структуру документа XML, чтобы иметь возможность выбирать индивидуальные элементы из нее. Хотя XPath может применяться к любому правильно оформленному документу XML, требование знать структуру документа при создании запроса означает, что действительность документа также гарантирует возможность работы запроса для различных документов, которые соответствуют той же самой схеме.

23

Введение в LINQ

В ЭТОЙ ГЛАВЕ...

- Кодирование запроса LINQ и частей оператора запроса LINQ
- Использование синтаксиса методов LINQ вместо синтаксиса запросов LINQ
- Применение лямбда-выражений в LINQ
- Упорядочивание результатов запроса, включая многоуровневое упорядочивание
- Когда и как следует использовать агрегатные операции LINQ
- Использование проекций для создания новых объектов в запросах
- Использование операций `Distinct()`, `Any()`, `All()`, `First()`, `FirstOrDefault()`, `Take()` и `Skip()`
- Групповые запросы
- Операции множеств и соединения

В этой главе представлен язык интегрированных запросов LINQ (Language Integrated Query) — новое расширение языка C#, которое появилось в версии C# 3.0, предшествовавшей C# 4 — язык, поддерживаемый в Visual Studio 2010. Язык LINQ решает проблему работы с очень большими коллекциями объектов, когда для решения определенной задачи обычно приходится выбирать подмножество коллекции.

В прошлом такого рода работа требовала написания объемного циклического кода и дополнительной обработки, связанной с сортировкой или группированием найденных объектов, которая порождала еще больший код. Язык LINQ избавляет от необходимости писать дополнительный циклический код для фильтрации и сортировки. Он позволяет сосредоточиться на объектах, которыми оперирует программа.

В дополнение к элегантному языку запросов, который позволяет в точности указывать искомые объекты, LINQ предлагает также множество методов расширения, которые облегчают задачу сортировки, группирования и вычисления статистики на результатах запросов.

С помощью LINQ можно опрашивать множество различных источников данных в C#, включая объекты, базы данных SQL, документы XML, сущностные модели данных и внешние приложения, такие как веб-службы Amazon и корпоративные каталоги. Синтаксис LINQ и методы, показанные в этой главе, одинаковы для всех источников данных. Поставщики LINQ для опроса различных источников рассматриваются в следующей главе.

Язык LINQ слишком обширен, чтобы можно было полностью раскрыть все его средства и методы; это вышло бы за рамки книги для начинающих. Однако здесь будут показаны примеры всех типов операций и операторов, которые, скорее всего, понадобятся вам как пользователю LINQ. Кроме того, будут представлены ссылки на ресурсы, необходимые для углубленного изучения этой темы.

Первый запрос LINQ

Начнем с примера. В следующем практическом занятии вы создадите запрос для поиска некоторых данных в простом, находящемся в памяти, массиве объектов с использованием LINQ и распечатаете результат на консоли.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Первая программа LINQ

Для создания примера выполните следующие шаги в Visual C# 2010.

1. Создайте новое консольное приложение по имени `23-01-FirstLINQquery` в каталоге `C:\BegVCSharp\Chapter23`, затем откройте главный файл `Program.cs`.
2. Обратите внимание, что Visual C# 2010 по умолчанию включает пространства имен LINQ в `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

3. Добавьте следующий код в метод `Main()` в `Program.cs`:

```
static void Main(string[] args)
    {
      string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
          "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
      var queryResults =
          from n in names
          where n.StartsWith("S")
          select n;
```

```
    Console.WriteLine("Names beginning with S:"); // Имена, начинающиеся с S:
    foreach (var item in queryResults) {
      Console.WriteLine(item);
    }
    Console.Write("Program finished, press Enter/Return to continue:");
    // Программа завершена, нажмите Enter для продолжения
    Console.ReadLine();
  }
```

*Фрагмент кода **23-01-FirstLINQquery\Program.cs***

4. Скомпилируйте и запустите программу (можно просто нажать <F5> для запуска отладки). Вы увидите имена в списке, начинающиеся с `S`, в том порядке, в каком они объявлены в массиве:

```
Names beginning with S:
Smith
Smythe
Small
Singh
Samba
Program finished, press Enter/Return to continue:
```

Просто нажмите <Enter> для завершения программы и закрытия экрана консоли. Если вы использовали <Ctrl+F5> (запуск без отладки), может понадобиться нажать <Enter> два раза. Это завершит выполнение программы.

Описание работы

Первый шаг — ссылка на пространство имен `System.Linq`, которая выполняется автоматически Visual C# 2010 при создании проекта:

```
using System.Linq;
```

Все базовые системные классы поддержки LINQ находятся в пространстве имен `System.Linq`. Если вы создаете исходный файл C# вне Visual C# 2010 или редактируете ранее существовавший проект Visual C# 2005, может понадобиться добавить эту директиву вручную.

Следующий шаг — подготовка некоторых данных, что в рассматриваемом примере осуществляется за счет объявления и инициализации массива `names`:

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
    "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
```

Это тривиальный набор данных, тем не менее, он подходит для начального примера, в котором результат запроса очевиден. Сам оператор запроса LINQ представлен в следующей части программы:

```
var queryResults =
    from n in names
    where n.StartsWith("S")
    select n;
```

Оператор выглядит довольно странно, не правда ли? Он кажется написанным на языке, отличном от C#, и в самом деле, синтаксис `from...where...select` определенно напоминает язык запросов баз данных SQL. Однако это не SQL! Это действительно C#, как вы убедились, набирая этот код в Visual C# 2010 — слова `from`, `where` и `select` подсвечивались как ключевые, и этот странно выглядящий синтаксис вполне нормально был воспринят компилятором.

Оператор запроса LINQ в этой программе использует декларативный синтаксис запросов LINQ:

```
var queryResults =
    from n in names
    where n.StartsWith("S")
    select n;
```

Этот оператор состоит из четырех частей: начинающееся с `var` объявление переменной `result`, которой выполняется присваивание с использованием *выражения запроса*, состоящего из конструкций `from`, `where` и `select`. Давайте рассмотрим каждую из этих частей по очереди.

Объявление переменной результата с использованием ключевого слова `var`

Запрос LINQ начинается с объявления переменной, которая будет содержать результат запроса, что обычно делается с применением ключевого слова `var`:

```
var queryResult =
```

Как было описано в главе 14, `var` — это новое ключевое слово в C# 3.0, предназначенное для объявления обобщенного типа переменных, что идеально для хранения результатов запросов LINQ. Ключевое слово `var` заставляет компилятор C# вывести тип результата на основе самого запроса. Таким образом, вам не нужно заранее объявлять тип объектов, возвращаемых запросом LINQ — компилятор позаботится об этом сам. Если запрос может вернуть множество элементов, то этот тип должен вести себя как коллекция объектов из источника данных запроса (формально это не коллекция; она лишь выглядит так).

Для тех, кого интересуют подробности: результат запроса будет типом, реализующим интерфейс `IEnumerable<>`. Угловые скобки (<>) после имени означают, что это обобщенный тип. Обобщения рассматриваются в главе 12. В данном конкретном случае компилятор создает экземпляр `System.Linq.OrderSequence<string, string>` — специального типа данных LINQ, который представляет упорядоченный список строк (потому что источником данных является коллекция строк).

Кстати, имя `queryResult` произвольно — вы можете назвать результат так, как вам нравится. Это может быть `namesBeginningWithS` или что-нибудь другое, что имеет смысл внутри программы.

Указание источника данных: конструкция `from`

Следующая часть запроса LINQ — конструкция `from`, которая указывает источник запрашиваемых данных:

```
from n in names
```

Источником данных в рассматриваемом случае является `names` — массив строк, объявленный ранее. Переменная `n` — просто заместитель индивидуальных элементов в источнике данных, подобно переменной `name`, следующей за оператором `foreach`. С помощью `from` вы указываете, что собираетесь запросить подмножество коллекции, а не выполнять итерацию по всем элементам.

Говоря об итерации, мы подразумеваем, что источник данных LINQ должен быть перечислимым — т.е. он должен быть массивом или коллекцией элементов, из которой их можно извлекать по одному.

Перечислимый источник — это такой источник данных, который поддерживает интерфейс `IEnumerable<>`, что присуще любому массиву или коллекции элементов C#.

Источник данных не может быть единственным значением или объектом, таким как отдельная переменная `int`. И в самом деле, если уже есть единственный элемент, зачем его запрашивать?

Указание условия: конструкция `where`

В следующей части запроса LINQ указывается условие запроса с использованием конструкции `where`, которая выглядит так:

```
where n.StartWith("S")
```

В конструкции `where` может быть специфицировано любое булевское выражение, которое применимо к элементам в источнике данных. На самом деле конструкция `where` не обязательна, и даже может быть опущена, но почти во всех случаях понадобится указывать условие `where`, чтобы ограничить результат только необходимыми данными. Конструкция `where` называется *ограничивающей операцией* в LINQ, потому что она ограничивает результат запроса.

Здесь определяется, что строка `name` должна начинаться с буквы `S`, но можно было бы задать и что-нибудь другое, например, что длина должна быть больше 10 (`where n.Length > 10`) либо имя должно содержать в себе букву `Q` (`where n.Contains("Q")`).

Выбор элементов: конструкция `select`

И, наконец, конструкция `select` указывает, какие элементы появятся в результирующем наборе. Выглядит она следующим образом:

```
select n
```

Конструкция `select` необходима, так как должны быть указаны элементы, которые запрос поместит в результирующий набор. Для рассматриваемого набора данных это не очень интересно, поскольку в каждом объекте результирующего набора есть только один элемент — имя. Далее будут приведены некоторые примеры более сложных объектов в результирующем наборе, где польза от конструкции `select` будет более наглядной, а пока давайте завершим первый пример.

Последний штрих: использование цикла `foreach`

Теперь результаты запроса должны быть выведены. Подобно массиву, использованному в качестве источника данных, результат запроса LINQ вроде этого является *перечислимым* — в том смысле, что по результату может быть выполнена итерация с помощью оператора `foreach`:

```
Console.WriteLine("Names beginning with S:");
foreach (var item in queryResults) {
      Console.WriteLine(item);
}
```

В этом случае получается совпадение для четырех имен — `Singh`, `Small`, `Smythe` и `Samba`, и именно они будут выведены в цикле `foreach`.

Отложенное выполнение запросов

Может показаться, что цикл `foreach` не является частью самого LINQ — ведь он просто перебирает полученные результаты. Хотя и верно, что конструкция `foreach` не принадлежит только LINQ, тем не менее, это часть кода, которая в действительности выполняет запрос LINQ! Присваивание результата запроса переменной лишь сохраняет план его выполнения; в LINQ сами данные не извлекаются до тех пор, пока не будет выполнено обращение к результату. Это называется *отложенным выполнением запросов*, или "ленивым"

выполнением. Выполнение откладывается для любого запроса, производящего последовательности, т.е. список результатов.

Теперь вернемся к коду. Поскольку результаты выведены, программу можно завершить:

```
Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
```

Эти строки лишь гарантируют, что результаты выполнения консольной программы останутся на экране до тех пор, пока не будет нажата клавиша <Enter>, даже если программа запускалась нажатием <F5> вместо <Ctrl+F5>. Такая конструкция будет применяться и в других примерах LINQ.

Использование синтаксиса методов LINQ

В LINQ доступно много способов решения одной и той же задачи, как это часто бывает в программировании. Как уже отмечалось, предыдущий пример написан с использованием *синтаксиса запросов* LINQ. В следующем примере та же программа будет реализована с применением *синтаксиса методов* LINQ (также называемого *явным синтаксисом*).

Методы расширения LINQ

Язык LINQ реализован в виде серии методов расширения для коллекций, массивов, результатов запросов и любых других объектов, реализующих интерфейс `IEnumerable`. Вы можете видеть эти методы в средстве IntelliSense Visual Studio. Например, откройте файл `Program.cs` только что скомпилированной программы `23-01-FirstLINQquery` в Visual C# 2010 и введите новую ссылку на тот же массив names, как показано ниже:

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
"Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
names.
```

Как только вы введете точку после `names`, то сразу увидите в раскрывающемся списке IntelliSense все доступные методы объекта `names`.

Метод `Where<>` и большинство прочих доступных методов являются методами расширения (что отмечено в документации словом `extension`, следующим за `Where<>`). Чтобы увидеть, что они являются расширениями LINQ, закомментируйте директиву `using System.Linq` в начале файла. Тогда `Where<>`, `Union<>`, `Takes<>` и большинство других методов исчезнут из этого списка. Выражение запроса `for...where...select`, использованное в предыдущем примере, транслируется компилятором C# в серию вызовов этих методов. При использовании синтаксиса методов LINQ эти методы вызываются напрямую.

Сравнение синтаксиса запросов и синтаксиса методов

Синтаксис запросов является предпочтительным способом программирования запросов в LINQ, поскольку его намного легче читать и проще использовать в большинстве запросов. Однако важно иметь базовое понятие о синтаксисе методов, поскольку некоторые возможности LINQ либо недоступны в синтаксисе запросов, либо просто их легче применять в синтаксисе методов.

Как рекомендует онлайновая справочная система Visual C# 2010, используйте синтаксис запросов, когда возможно, и синтаксис методов, когда необходимо.

В этой главе применяется в основном синтаксис запросов, но будут показаны ситуации, когда необходим синтаксис методов, и продемонстрировано, как использовать синтаксис методов для решения проблемы.

Большинство методов LINQ, использующих синтаксис методов, требуют передачи метода или функции для вычисления выражения запроса. Параметр метода/функции передается в форме делегата, который часто называют анонимным методом.

К счастью, LINQ позволяет сделать это намного проще, чем можно было предположить! Вы создаете метод/функцию, используя лямбда-выражением (lambda expression), как описано в главе 14.

Для более ясного представления рассмотрим реальное применение этого.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование синтаксиса методов LINQ

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Либо модифицируйте пример `23-01-FirstLINQquery`, либо создайте новое консольное приложение по имени `23-02-LINQMethodSyntax` в каталоге `C:\BegVCSharp\Chapter23`. Откройте главный файл `Program.cs`.

2. Visual C# 2010 автоматически включает нужное пространство имен в `Program.cs`:

```
using System.Linq;
```

3. Добавьте следующий код в метод `Main()` внутри `Program.cs`:

```
static void Main(string[] args)
    {
      string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
          "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
      var queryResults = names.Where(n => n.StartsWith("S"));
      Console.WriteLine("Names beginning with S:");
      foreach (var item in queryResults) {
        Console.WriteLine(item);
      }
      Console.Write("Program finished, press Enter/Return to continue:");
      Console.ReadLine();
    }
```

*Фрагмент кода **23-02-LINQMethodSyntax\Program.cs***

4. Скомпилируйте и запустите эту программу (можно просто нажать <F5>). Вы увидите тот же вывод списка имен, начинающихся с `S`, в порядке, в котором они объявлены в массиве:

```
Names beginning with S:
Smith
Smythe
Small
Singh
Samba
Program finished, press Enter/Return to continue:
```

Описание работы

Как и ранее, ссылка на пространство имен `System.Linq` вставляется Visual C# 2010 автоматически:

```
using System.Linq;
```

Те же исходные данные, что и ранее, создаются вновь с помощью объявления и инициализации массива имен:

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
   "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
```

Отличающаяся часть — запрос LINQ, который теперь вызывает метод `Where()` вместо выражения запроса:

```
var queryResults = names.Where(n => n.StartsWith("S"));
```

Компилятор C# превращает лямбда-выражение `n => n.StartsWith("S")` в анонимный метод, выполняемый `Where()` на каждом элементе массива `names`. Если лямбда-выражение возвращает для элемента `true`, он включается в результат, возвращаемый `Where()`. Компилятор C# определяет, что `Where()` должен принимать `string` в качестве входного типа для каждого элемента входного источника (в данном случае — массива `names`).

Как много происходит в единственной строке кода, не правда ли? Для такого простого типа запроса синтаксис метода существенно короче, чем синтаксис запроса, потому что конструкции `from` и `select` не нужны; однако большинство запросов, с которыми вам придется иметь дело, сложнее этого.

Остальная часть примера та же, что и в предыдущем случае — вы просто выводите результаты запроса в цикле `foreach` и приостанавливаете вывод в конце, чтобы можно было увидеть его перед тем, как программа завершится:

```
foreach (var item in queryResults) {
  Console.WriteLine(item);
}
Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
```

Мы не станем повторять здесь объяснения этих строк, поскольку оно уже было приведено в разделе "Описание работы" после первого примера этой главы. Теперь давайте двинемся дальше и рассмотрим другие возможности LINQ.

Упорядочивание результатов запроса

Как только вы нашли интересующие данные в конструкции `where` (или в вызове метода `Where()`), LINQ облегчает дальнейшую обработку, такую как сортировка результирующих данных. В следующем практическом занятии результаты первого запроса будут представлены в алфавитном порядке.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Упорядочение результатов запроса

Выполните следующие шаги для создания нового примера в Visual C# 2010.

1. Либо модифицируйте пример `23-01-FirstLINQquery`, либо создайте новый проект консольного приложения по имени `23-03-OrderQueryResults` в каталоге `C:\BegVCSharp\Chapter23`.
2. Откройте главный файл `Program.cs`. Как и ранее, Visual C# 2010 автоматически включает директиву `using System.Linq;` в `Program.cs`.
3. Добавьте следующий код в метод `Main()` внутри `Program.cs`:

```
static void Main(string[] args)
    {
      string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
                         "Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich",
                         "Singh", "Samba", "Fatimah" };
      var queryResults =
        from n in names
        where n.StartsWith("S")
        orderby n
        select n;
```

```
    Console.WriteLine("Names beginning with S ordered alphabetically:");
    foreach (var item in queryResults) {
      Console.WriteLine(item);
    }
    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
  }
```

Фрагмент кода ***23-03-OrderQueryResults\Program.cs***

4. Скомпилируйте и выполните программу. Вы увидите имена, начинающиеся с буквы "S", в алфавитном порядке:

```
Names beginning with S ordered alphabetically:
Samba
Singh
Small
Smith
Smythe
Program finished, press Enter/Return to continue:
```

Описание работы

Эта программа почти идентична предыдущему примеру, за исключением одной дополнительной строки, добавленной к оператору запроса:

```
var queryResults =
  from n in names
  where n.StartsWith("S")
  orderby n
  select n;
```

Конструкция `orderby`

Конструкция `orderby` выглядит следующим образом:

```
orderby n
```

Подобно `where`, конструкция `orderby` необязательна. Добавив всего одну строку, вы можете выстроить в нужном порядке результаты любого запроса, что в противном случае потребовало бы нескольких строк дополнительного кода и, возможно, дополнительных методов или коллекций для хранения результатов, упорядоченных различным образом в зависимости от алгоритма сортировки, который вы предпочтете реализовать. При наличии нескольких типов, которые нужно упорядочить, пришлось бы реализовать множество методов упорядочивания для каждого из них. Благодаря LINQ, больше не нужно беспокоиться об этом: просто добавьте дополнительную конструкцию к оператору запроса.

По умолчанию `orderby` упорядочивает элементы результата по возрастанию (от A до Z), однако можно указать и порядок по убыванию (от Z до A), просто добавив ключевое слово `descending`:

```
orderby n descending
```

Это упорядочит результат примера следующим образом:

```
Smythe
Smith
Small
Singh
Samba
```

Кроме того, можно упорядочивать по произвольному выражению, не переписывая запрос. Например, чтобы упорядочить по последней букве в имени вместо нормального алфавитного порядка, вы просто изменяете конструкцию `orderby` следующим образом:

```
orderby n.Substring(n.Length - 1)
```

Результат показан ниже:

```
Samba
Smythe
Smith
Singh
Small
```

Последние буквы выстроены по алфавиту (`a`, `e`, `h`, `h`, `l`). Однако вы заметите, что выполнение зависит от реализации, в том смысле, что нет никаких гарантий относительно порядка, помимо указанного в конструкции `orderby`. Первая буква является единственной учитываемой, поэтому в данном случае "Smith" располагается перед "Singh".

Упорядочивание с использованием синтаксиса методов

Чтобы добавить к запросу такие возможности, как упорядочивание результата, просто добавьте вызов метода для каждой операции LINQ, которую хотите выполнить в своем запросе LINQ на основе методов. Опять-таки, это проще, чем может показаться на первый взгляд, что демонстрируется в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Упорядочение с использованием синтаксиса методов**

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Либо модифицируйте пример `23-02-LINQMethodSyntax`, либо создайте новый проект консольного приложения по имени `23-04-OrderMethodSyntax` в каталоге `C:\BegVCSharp\Chapter23`.

2. Добавьте следующий код в метод `Main()` в `Program.cs`. Как и во всех примерах Visual C# 2010, он автоматически включит ссылку на пространство имен `System.Linq`.

```
static void Main(string[] args)
    {
      string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
          "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
      var queryResults = names.OrderBy(n => n).Where(n => n.StartsWith("S"));
      Console.WriteLine("Names beginning with S:");
      foreach (var item in queryResults) {
        Console.WriteLine(item);
      }
      Console.Write("Program finished, press Enter/Return to continue:");
      Console.ReadLine();
    }
```

*Фрагмент кода **`23-04-OrderMethodSyntax\Program.cs`***

3. Скомпилируйте и запустите программу. Вы увидите список имен, начинающихся с буквы "S", в алфавитном порядке — точно так де, как и в выводе предыдущего примера.

Описание работы

Этот пример почти идентичен предыдущему примеру синтаксиса метода, за исключением добавления вызова метода LINQ по имени `QueryBy()`, предшествующего вызову метода `Where()`:

```
var queryResults = names.OrderBy(n => n).Where(n => n.StartsWith("S"));
```

Как вы, возможно, заметили в IntelliSense, набирая код в редакторе, метод `OrderBy()` возвращает `IOrderedEnumerable`, который является надмножеством интерфейса `IEnumerable`, поэтому можно вызывать `Where()` на нем точно так же, как это делается с любым другим объектом `IEnumerable`.

Компилятор делает вывод, что вы работаете с данными `string`, поэтому типы данных появляются в IntelliSense как `IOrderedEnumerable<string>` и `IEnumerable<string>`.

Вы должны передать лямбда-выражение методу `OrderBy()`, чтобы сообщить ему, какую функцию следует применять для упорядочивания. Вы передаете простейшее возможное лямбда-выражение `n => n`, потому что не нуждаетесь в упорядочивании ничего другого, кроме самого выводимого значения. В синтаксисе запросов создавать это дополнительное лямбда-выражение не нужно.

Чтобы упорядочить элементы в обратном порядке, необходимо вызвать метод `OrderByDescending()`:

```
var queryResults = names.OrderByDescending(n => n).Where(n =>
n.StartsWith("S"));
```

Это даст те же результаты, что и конструкция `orderby n descending`, которая применялась в версии с синтаксисом запросов.

Если нужно упорядочить по чему-нибудь, отличному от самого значения, можно изменить лямбда-выражение, передаваемое `OrderBy()`. Например, чтобы упорядочить по последней букве каждого имени, необходимо использовать следующее лямбда-выражение: `n => n.Substring(n.Length - 1)` и передать его `OrderBy`, как показано ниже:

```
var queryResults =
   names.OrderBy(n => n.Substring(n.Length-1)).Where(n => n.StartsWith("S"));
```

Это даст тот же результат, упорядоченный по последней букве имени, как и в предыдущем примере.

Организация запросов к большим наборам данных

Вы можете сказать, что в синтаксисе LINQ все прекрасно, но в чем смысл всего этого? Вы можете и без него замечательно увидеть нужный результат, просто взглянув на исходный массив, так зачем же морочить голову, выстраивая запросы того, что и так очевидно? Как упоминалось ранее, иногда результаты запроса не столь очевидны. В следующем практическом занятии создается огромный массив чисел, которому выполняются запрос с помощью LINQ.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Упорядочение большого набора данных**

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новое консольное приложение по имени 23-05-LargeNumberQuery в каталоге C:\BegVCSharp\Chapter23. Как и ранее, при создании этого проекта Visual C# 2010 включит необходимое пространство имен LINQ в файл Program.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

2. Добавьте следующий код в метод Main():

```
static void Main(string[] args)
{
  int[] numbers = generateLotsOfNumbers(12345678);

  var queryResults =
      from n in numbers
      where n < 1000
      select n
      ;

  Console.WriteLine("Numbers less than 1000:"); // Числа меньше 1000
  foreach (var item in queryResults)
  {
    Console.WriteLine(item);
  }
  Console.Write("Program finished, press Enter/Return to continue:");
  Console.ReadLine();
}
```

*Фрагмент кода **23-05-LargeNumberQuery\Program.cs***

3. Добавьте следующий метод для генерации списка случайных чисел:

```
private static int[] generateLotsOfNumbers(int count)
{
  Random generator = new Random(0);
  int[] result = new int[count];
  for (int i = 0; i < count; i++)
  {
    result[i] = generator.Next();
  }
  return result;
}
```

4. Скомпилируйте и запустите программу. Вы увидите список чисел меньше 1000, как показано ниже:

```
Numbers less than 1000:
714
24
677
350
257
719
584
Program finished, press Enter/Return to continue:
```

Описание работы

Как и ранее, первый шаг — это ссылка на пространство имен System.Linq, которая добавляется автоматически Visual C# 2010 при создании проекта:

```
using System.Linq;
```

Следующий шаг — генерация некоторых данных, которая осуществляется в этом примере вызовом метода `generateLotsOfNumbers()`:

```
int[] numbers = generateLotsOfNumbers(12345678);
private static int[] generateLotsOfNumbers(int count)
{
  Random generator = new Random(0);
  int[] result = new int[count];
  for (int i = 0; i < count; i++)
  {
    result[i] = generator.Next();
  }
  return result;
}
```

Это не простой набор данных — в массиве свыше 12 миллионов чисел! В одном из упражнений в конце этой главы необходимо будет изменить параметр `size`, передаваемый методу `generateLotsOfNumbers()` для генерации произвольного размера наборов случайных чисел, чтобы посмотреть, как это влияет на результаты запроса. Как вы увидите при выполнении упражнения, указанный здесь размер, 12 345 678, достаточно велик для получения некоторых случайных чисел меньше 1000, чтобы получить результаты для демонстрации первого запроса.

Значения должны быть равномерно распределены по диапазону целых со знаком (от нуля до более чем 2 миллиарда). Создав генератор случайных чисел с начальным значением `0`, вы гарантируете генерацию одной и той же последовательности случайных чисел каждый раз, так что получите один и тот же результат запроса, показанный здесь, но каким именно будет этот результат — неизвестно до момента выполнения запроса. К счастью, LINQ существенно упрощает решение этой задачи!

Сам оператор запроса подобен тому, который выполнялся ранее с именами, и состоит в том, чтобы выбрать некоторые числа, отвечающие определенному условию (в данном случае — числа меньше 1000):

```
var queryResults =
   from n in numbers
   where n < 1000
   select n
```

Конструкция `orderby` на этот раз не нужна и потребовала бы дополнительного времени на обработку (не слишком существенного в данном примере, но заметного, когда вы измените запрос в следующем примере).

Результаты запроса выводятся в операторе `foreach`, как и в предыдущем примере:

```
Console.WriteLine("Numbers less than 1000:");
foreach (var item in queryResults) {
     Console.WriteLine(item);
}
```

Опять-таки, здесь присутствует вывод на консоль и чтение символа для приостановки вывода:

```
Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
```

Код паузы во всех последующий примерах один и тот же, поэтому мы не станем показывать его вновь и вновь.

В LINQ очень легко изменить условия запроса для представления различных характеристик набора данных. Однако в зависимости от того, насколько много результатов возвращает запрос, может оказаться бессмысленным вывод их всех каждый раз. В следующем разделе будет показано, какие агрегатные операции предлагает LINQ, чтобы справиться с этой проблемой.

Агрегатные операции

Часто запрос возвращает больше результатов, чем ожидалось. Например, если вы измените условие в предыдущей программе запроса чисел, указав, что необходимы числа больше 1000, а не меньше 1000, результат окажется просто нескончаемым, и вам захочется попросту прервать вывод.

К счастью, LINQ предлагает набор агрегатных операций, позволяющих анализировать результаты запроса, не выполняя цикл по всем результатам. Перечисленные в табл. 23.1 агрегатные операции чаще всего используются для набора числовых результатов, таких как результаты предыдущего запроса, и могут быть знакомы, если вы имели дело с таким языком запросов баз данных, как SQL.

Таблица 23.1. Агрегатные операции

Операция	Описание
`Count()`	Количество результатов
`Min()`	Минимальное значение результатов
`Max()`	Максимальное значение результатов
`Average()`	Среднее значение числовых результатов
`Sum()`	Сумма всех числовых результатов

Существуют и другие агрегатные операции, такие как `Aggregate()`, которые предназначены для выполнения произвольного кода в манере, позволяющей самостоятельно кодировать агрегатную функцию. Однако это — тема для профессиональных разработчиков, и она выходит за рамки настоящей книги.

Поскольку агрегатные операции возвращают простой скалярный тип вместо последовательности результатов, их применение вызывает немедленное выполнение всего запроса вместо обычного отложенного выполнения запроса.

В следующем практическом занятии вы модифицируете большой числовой запрос из предыдущего примера и применяете агрегатные операции для исследования результирующего набора из версии “больше чем” этого запроса, используя LINQ.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Числовые агрегатные операции

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Либо модифицируйте пример `23-05-LargeNumberQuery`, либо создайте новый проект консольного приложения по имени `23-06-NumericAggregates` в каталоге `C:\BegVCSharp\Chapter23`.
2. Как и ранее, при создании проекта Visual C# 2010 включает в `Program.cs` ссылку на пространство имен LINQ. Вам нужно лишь модифицировать метод `Main()`, как показано в следующем коде и в остальной части этого практического занятия. Как и в предыдущем примере, в этом запросе не используется конструкция `orderby`. Однако условие конструкции `where` противоположно предыдущему примеру (числа должны быть больше 1000, а не меньше, как ранее).

```
static void Main(string[] args)
{
  int[] numbers = generateLotsOfNumbers(12345678);
  Console.WriteLine("Numeric Aggregates"); // числовые агрегаты
  var queryResults =
      from n in numbers
      where n > 1000
      select n
      ;
  Console.WriteLine("Count of Numbers > 1000");
                  // Количество чисел > 1000
  Console.WriteLine(queryResults.Count());
  Console.WriteLine("Max of Numbers > 1000");
                  // Максимальное из чисел > 1000
  Console.WriteLine(queryResults.Max());
  Console.WriteLine("Min of Numbers > 1000");
                  // Минимальное из чисел > 1000
  Console.WriteLine(queryResults.Min());
  Console.WriteLine("Average of Numbers > 1000");
                  // Среднее из чисел > 1000
  Console.WriteLine(queryResults.Average());
  Console.WriteLine("Sum of Numbers > 1000");
                  // Сумма чисел > 1000
  Console.WriteLine(queryResults.Sum(n => (long) n));
  Console.Write("Program finished, press Enter/Return to continue:");
  Console.ReadLine();
}
```

*Фрагмент кода **23-06-NumericAggregates\Program.cs***

3. Если его еще нет, добавьте метод `generateLotsOfNumbers()`, использованный в предыдущем примере:

```
private static int[] generateLotsOfNumbers(int count)
{
  Random generator = new Random(0);
  int[] result = new int[count];
  for (int i = 0; i < count; i++)
  {
    result[i] = generator.Next();
  }
  return result;
}
```

4. Скомпилируйте и выполните пример. Вы увидите значения количества чисел, удовлетворяющих условию `where`, минимальное, максимальное и среднее их значения, как показано ниже:

```
Numeric Aggregates
Count of Numbers > 1000
12345671
Max of Numbers > 1000
2147483591
Min of Numbers > 1000
1034
Average of Numbers > 1000
1073643807.50298
Sum of Numbers > 1000
13254853218619179
Program finished, press Enter/Return to continue:
```

Этот запрос производит намного больше результатов, чем в предыдущем примере (более 12 миллионов). Использование `orderby` на таком большом результирующем наборе определенно пагубно скажется на производительности! Самое большое число в результирующем наборе превышает значение в 2 миллиарда, а самое маленькое — чуть больше 1000, как и ожидалось. Среднее — около 1 миллиона — как раз находится около середины диапазона возможных значений. Похоже, функция `Rand()` генерирует равномерное распределение чисел.

Описание работы

Первая часть программы точно такая же, как и в предыдущем примере, со ссылкой на пространство имен `System.Linq` и использованием метода `generateLotsOfNumbers()` для генерации исходных данных:

```
int[] numbers = generateLotsOfNumbers(12345678);
```

Запрос — тот же самый, что и в предыдущем примере, за исключением того, что условие `where` изменено с "меньше чем" на "больше чем":

```
var queryResults =
    from n in numbers
    where n > 1000
    select n;
```

Как упоминалось ранее, запрос, использующий условие "больше чем", производит намного больше результатов, чем запрос с условием "меньше чем" (на этом конкретном наборе данных). Используя агрегатные операции, вы можете исследовать результаты запроса, не выводя на консоль каждый элемент и не сравнивая его в цикле `foreach`. Каждая операция выступает в виде метода, вызываемого на результирующем наборе, подобно методам типа коллекции.

Рассмотрим применение каждой агрегатной операции.

- **`Count()`:**

  ```
  Console.WriteLine("Count of Numbers > 1000");
  Console.WriteLine(queryResults.Count());
  ```

  `Count()` возвращает количество строк в результате запроса, в данном случае — 12 345 671 строк.

- **`Max()`:**

  ```
  Console.WriteLine("Max of Numbers > 1000");
  Console.WriteLine(queryResults.Max());
  ```

  `Max()` возвращает максимальное значение в результатах запроса, в данном случае — число больше 2 миллиардов: 2 147 483 591, что очень близко к максимальному значению типа `int` (`int.MaxValue`, или 2 147 483 647).

- **`Min()`:**

  ```
  Console.WriteLine("Min of Numbers > 1000");
  Console.WriteLine(queryResults.Min());
  ```

  `Min()` возвращает минимальное значение в результатах запроса, в данном случае — 1034.

- **`Average()`:**

  ```
  Console.WriteLine("Average of Numbers > 1000");
  Console.WriteLine(queryResults.Average());
  ```

  `Average()` возвращает среднее значение из результатов запроса, которым в данном случае является `1073643807.50298` — значение, очень близкое к середине диапазона возможных значений от 1000 до более 2 миллиардов. Это довольно бессмысленно для произвольного набора чисел, но демонстрирует возможности анализа

результатов запроса. В последней части этой главы мы рассмотрим более полезное применение этих операций к некоторым бизнес-данным.

➤ **Sum():**

```
Console.WriteLine("Sum of Numbers > 1000");
Console.WriteLine(queryResults.Sum(n => (long) n));
```

Обратите внимание, что вы передали лямбда-выражение `n => (long) n` методу `Sum()` для того, чтобы получить сумму всех чисел. Хотя `Sum()` не имеет перегрузок с другими параметрами, как `Count()`, `Min()`, `Max()` и т.д., использование этой версии вызова метода может привести к ошибке переполнения, потому что в результирующем наборе настолько много больших чисел, что их сумма может превысить значение, умещающееся в 32-битное целое, которое возвращает версия `Sum()` без параметров. Лямбда-выражение позволяет преобразовать результат `Sum()` в длинное 64-битное целое, которое нужно для хранения значения, превышающее 13 квадриллионов, не вызывая переполнения, а именно — 13 254 853 218 619 179, и лямбда-выражение позволяет легко выполнить такого рода поправку.

В дополнение к методу `Count()`, который возвращает 32-битное целое, LINQ также предусматривает метод `LongCount()`, возвращающий результат запроса в виде 64-битного целого. Однако это — специальный случай; все прочие операции требуют специального лямбда-выражения или вызова метода преобразования, если необходима 64-битная версия числа.

Запросы сложных объектов

В предыдущих примерах демонстрировалось, как запросы LINQ могут работать со списками простых типов, таких как числа и строки. Теперь будет показано, как использовать запросы LINQ с более сложными объектами. Вы создадите простой класс `Customer`, включающий достаточно информации для испытания некоторых интересных запросов.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Запрос сложных объектов

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новое консольное приложение по имени `23-07-QueryComplexObjects` в каталоге `C:\BegVCSharp\Chapter23`.
2. Перед классом `Program` в `Program.cs` добавьте показанное ниже краткое определение класса `Customer`:

```
class Customer
    {
      public string ID { get; set; }
      public string City { get; set; }
      public string Country { get; set; }
      public string Region { get; set; }
      public decimal Sales { get; set; }
      public override string ToString()
      {
        return "ID: " + ID + " City: " + City + " Country: " + Country +
              " Region: " + Region + " Sales: " + Sales;
        // return "ID: " + ID + " Город: " + City + " Страна: " + Country +
        //       " Регион: " + Region + " Продажи: " + Sales;
      }
    }
```

*Фрагмент кода **23-07-QueryComplexObjects\Program.cs***

3. Добавьте следующий код в метод `Main()` класса `Program` внутри `Program.cs`:

```
static void Main(string[] args)
{
  List<Customer> customers = new List<Customer> {
     new Customer { ID="A", City="New York", Country="USA",
                 Region="North America", Sales=9999},
     new Customer { ID="B", City="Mumbai", Country="India",
                 Region="Asia", Sales=8888 },
     new Customer { ID="C", City="Karachi", Country="Pakistan",
                 Region="Asia", Sales=7777 },
     new Customer { ID="D", City="Delhi", Country="India",
                 Region="Asia", Sales=6666 },
     new Customer { ID="E", City="S г o Paulo", Country="Brazil",
                 Region="South America", Sales=5555 },
     new Customer { ID="F", City="Moscow", Country="Russia",
                 Region="Europe", Sales=4444 },
     new Customer { ID="G", City="Seoul", Country="Korea", Region="Asia",
                 Sales=3333 },
     new Customer { ID="H", City="Istanbul", Country="Turkey",
                 Region="Asia", Sales=2222 },
     new Customer { ID="I", City="Shanghai", Country="China", Region="Asia",
                 Sales=1111 },
     new Customer { ID="J", City="Lagos", Country="Nigeria",
                 Region="Africa", Sales=1000 },
     new Customer { ID="K", City="Mexico City", Country="Mexico",
                 Region="North America", Sales=2000 },
     new Customer { ID="L", City="Jakarta", Country="Indonesia",
                 Region="Asia", Sales=3000 },
     new Customer { ID="M", City="Tokyo", Country="Japan",
                 Region="Asia", Sales=4000 },
     new Customer { ID="N", City="Los Angeles", Country="USA",
                 Region="North America", Sales=5000 },
     new Customer { ID="O", City="Cairo", Country="Egypt",
                 Region="Africa", Sales=6000 },
     new Customer { ID="P", City="Tehran", Country="Iran",
                 Region="Asia", Sales=7000 },
     new Customer { ID="Q", City="London", Country="UK",
                 Region="Europe", Sales=8000 },
     new Customer { ID="R", City="Beijing", Country="China",
                 Region="Asia", Sales=9000 },
     new Customer { ID="S", City="Bogot б ", Country="Colombia",
                 Region="South America", Sales=1001 },
     new Customer { ID="T", City="Lima", Country="Peru",
                 Region="South America", Sales=2002 }
  };
  var queryResults =
     from c in customers
     where c.Region == "Asia"
     select c
     ;
  Console.WriteLine("Customers in Asia:"); // Заказчики в Азии
  foreach (Customer c in queryResults)
  {
    Console.WriteLine(c);
  }
  Console.Write("Program finished, press Enter/Return to continue:");
  Console.ReadLine();
 }
}
```

*Фрагмент кода **23-07-QueryComplexObjects\Program.cs***

4. Скомпилируйте и запустите программу. Ниже показан результирующий список клиентов из Азии:

```
Customers in Asia:
ID: B City: Mumbai Country: India Region: Asia Sales: 8888
ID: C City: Karachi Country: Pakistan Region: Asia Sales: 7777
ID: D City: Delhi Country: India Region: Asia Sales: 6666
ID: G City: Seoul Country: Korea Region: Asia Sales: 3333
ID: H City: Istanbul Country: Turkey Region: Asia Sales: 2222
ID: I City: Shanghai Country: China Region: Asia Sales: 1111
ID: L City: Jakarta Country: Indonesia Region: Asia Sales: 3000
ID: M City: Tokyo Country: Japan Region: Asia Sales: 4000
ID: P City: Tehran Country: Iran Region: Asia Sales: 7000
ID: R City: Beijing Country: China Region: Asia Sales: 9000
Program finished, press Enter/Return to continue:
```

Описание работы

В определении класса `Customer` используется средство автоматических свойств C# при объявлении общедоступных свойств (`ID`, `City`, `Country`, `Region`, `Sales`) класса `Customer`, не прибегая к явному кодированию приватных переменных экземпляра и кода `set/get` для каждого свойства:

```
class Customer
  {
    public string ID { get; set; }
    public string City { get; set; }
    ...
```

Единственный дополнительный метод, который придется закодировать в классе `Customer` — это переопределение метода `ToString()`, выдающего строковое представление для экземпляра `Customer`:

```
public override string ToString()
{
    return "ID: " + ID + " City: " + City + " Country: " + Country +
           " Region: " + Region + " Sales: " + Sales;
}
```

Метод `ToString()` используется для упрощения вывода результатов запроса.

В методе `Main()` класса `Program` создается строго типизированная коллекция типа `Customer` с использованием синтаксиса инициализации коллекций, чтобы избежать необходимости кодировать метод-конструктор и вызывать конструктор для создания каждого члена списка:

```
List<Customer> customers = new List<Customer> {
    new Customer { ID="A", City="New York", Country="USA",
                           Region="North America", Sales=9999},
    new Customer { ID="B", City="Mumbai", Country="India",
                           Region="Asia", Sales=8888 },
    ...
```

Ваши заказчики разбросаны по всему миру, и в данных присутствует достаточный объем географической информации, чтобы сформулировать интересные критерии выборки и группирования запросов.

По-прежнему в методе `Main()` создается оператор запроса — в данном случае для выборки заказчиков из Азии:

```
var queryResults =
    from c in customers
    where c.Region == "Asia"
    select c
    ;
```

Приведенный выше запрос должен быть хорошо знаком: это тот же LINQ-запрос `from...where...select`, который применялся в других примерах, но с тем отличием, что каждый элемент в результирующем списке представляет собой полноценный объект (`Customer`), а не простую строку или целое число. Затем в цикле `foreach` выводятся результаты:

```
Console.WriteLine("Customers in Asia:");
foreach (Customer c in queryResults)
{
   Console.WriteLine(c);
}
```

Цикл `foreach` здесь слегка отличается от аналогичных циклов в предыдущих примерах. Поскольку известно, что запрашиваются объекты `Customer`, переменная итерации явно объявляется, как относящаяся к типу `Customer`:

```
foreach (Customer c in queryResults)
```

Переменную `c` можно было бы объявить с ключевым словом `var`, и компилятор вывел бы тип переменной итерации — `Customer`, но явное объявление делает код более понятным для читателя-человека.

Внутри самого цикла вы просто пишете

```
{
  Console.WriteLine(c);
}
```

вместо явной печати полей `Customer`, потому что в классе `Customer` переопределен метод `ToString()`. Без переопределения метод `ToString()` по умолчанию просто выводил бы имя типа, как показано ниже:

```
Customers in Asia:
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
BegVCSharp_23_7_QueryComplexObjects.Customer
Program finished, press Enter/Return to continue:
```

Это совсем не то, что нужно! Разумеется, можно было бы явно вывести необходимые свойства `Customer`:

```
Console.WriteLine("Customer {0}: {1}, {2}", c.ID, c.City, c.Country);
```

Однако если интересует только несколько свойств объекта, было бы неэффективно извлекать в запросе весь объект целиком. К счастью, LINQ позволяет просто создать результаты запроса, которые содержат только нужные элементы — через проекцию, с которой мы поэкспериментируем в следующем разделе.

Проекция: создание новых объектов в запросах

Проекция означает создание новых типов данных из других типов данных в запросе LINQ. Ключевое слово `select` представляет собой операцию проекции, которая использовалась в предшествующих примерах. Если вы знакомы с ключевым словом `SELECT` в языке

запросов SQL, то вы уже знакомы и с операцией выбора определенного поля из объекта данных, в противоположность выбору всего объекта целиком. В LINQ также можно это делать; например, чтобы выбрать только поле `City` из списка `Customer` в предыдущем примере, просто измените конструкцию `select` в операторе запроса, чтобы она ссылалась только на свойство `City`:

```
var queryResults =
    from c in customers
    where c.Region == "Asia"
    select c.City
    ;
```

Это произведет следующий вывод:

```
Mumbai
Karachi
Delhi
Seoul
Istanbul
Shanghai
Jakarta
Tokyo
Tehran
Beijing
```

Можно даже трансформировать данные в запросе, добавив выражение в `select`, как показано ниже для числового типа данных:

```
select n + 1
```

или как показано здесь для запроса строкового типа данных:

```
select s.ToUpper()
```

Однако, в отличие от SQL, LINQ не допускает множественные поля в конструкции `select`. Это значит, что следующая строка:

```
select c.City, c.Country, c.Sales
```

вызовет ошибку компиляции (ожидается точка с запятой), потому что `select` принимает только один элемент в своем списке параметров.

Вместо этого в LINQ "на лету" создается новый объект в конструкции `select` для хранения результатов, которые нужны для запроса. Сделаем это в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Проекция: создание новых объектов в запросах

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Модифицируйте пример `23-07-QueryComplexObjects` или создайте новое консольное приложение по имени `23-08-ProjectionCreateNewObjects` в каталоге `C:\BegVCSharp\Chapter23`.
2. В случае нового проекта скопируйте код для создания класса `Customer` и инициализации списка заказчиков из примера `23-07-QueryComplexObjects`; этот код в точности повторяет ранее использованный.
3. В методе `Main()` выполните инициализацию списка `customers`, введите (или модифицируйте) запрос и цикл обработки результатов, как показано ниже:

```
var queryResults =
    from c in customers
    where c.Region == "North America"
```

```
        select new { c.City, c.Country, c.Sales }
        ;
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }
```

*Фрагмент кода **23-08-ProjectionCreateNewObjects\Program.cs***

4. Остальной код в методе `Main()` такой же, как и в предыдущих примерах.
5. Скомпилируйте и запустите программу. Вы увидите выбранные поля списка заказчиков из Северной Америки, перечисленные следующим образом:

```
{ City = New York, Country = USA, Sales = 9999 }
{ City = Mexico City, Country = Mexico, Sales = 2000 }
{ City = Los Angeles, Country = USA, Sales = 5000 }
Program finished, press Enter/Return to continue:
```

Описание работы

Класс `Customer` и список `customers` — те же, что и в предыдущем примере. Для разнообразия в запросе был изменен регион на Северную Америку. Интересное изменение, касающееся проекции, заключено в параметре конструкции `select`:

```
select new { c.City, c.Country, c.Sales }
```

Непосредственно в конструкции `select` используется синтаксис анонимного типа C# для создания нового объекта анонимного типа, имеющего свойства `City`, `Country` и `Sales`. Конструкция `select` создает новый объект. Таким образом, только эти три свойства дублируются и обрабатываются на разных стадиях обработки запроса.

При печати результатов запроса используется тот же обобщенный код цикла `foreach`, который применялся в предыдущих примерах, за исключением запроса `Customer`:

```
foreach (var item in queryResults)
{
  Console.WriteLine(item);
}
```

Этот код полностью обобщен; компилятор выводит тип результата запроса и вызывает правильные методы для анонимного типа, при этом явно кодировать что-либо не требуется. Не нужно даже переопределять `ToString()`, поскольку компилятор предоставляет реализацию `ToString()` по умолчанию, которая печатает имена свойств и значений в манере, подобной инициализации объекта.

Проекция: синтаксис методов

Версия в синтаксисе методов запроса проекции создается связыванием в цепочку вызова метода LINQ по имени `Select()` и других вызываемых методов LINQ. Например, тот же результат запроса можно получить, если добавить вызов метода `Select()` к вызову `Where()`, как показано ниже:

```
var queryResults = customers.Where(c => c.Region == "North America")
                        .Select(c => new { c.City, c.Country, c.Sales });
```

Хотя конструкция `select` обязательна в синтаксисе запросов, вы еще не видели ранее вызова метода `Select()`, потому что в синтаксисе методов LINQ он обязательным не является, если только не выполняется проекция (изменение типа результирующего набора по сравнению с исходным запрашиваемым типом).

Порядок вызовов методов не фиксирован, потому что все типы возврата методов LINQ реализуют интерфейс `IEnumerable` — можно вызвать `Select()` на результате `Where()`

и наоборот. Однако, в зависимости от специфики запроса, порядок может быть важен. Например, можно было бы поменять порядок вызовов `Select()` и `Where()` следующим образом:

```
var queryResults = customers.Select(c => new { c.City, c.Country, c.Sales })
                        .Where(c => c.Region == "North America");
```

Свойство `Region` не включено в анонимный тип `{ c.City, c.Country, c.Sales }`, созданный проекцией `Select()`, поэтому в методе `Where()` возникнет ошибка компиляции, указывающая на то, что анонимный тип не содержит определения `Region`.

Однако если бы метод `Where()` был ограничен данными, основанными на поле, которое включено в анонимный тип, такое как `City`, то проблем бы не было. Например, следующий запрос успешно компилируется и выполняется:

```
var queryResults = customers.Select(c => new {c.City, c.Country, c.Sales })
                        .Where(c => c.City == "New York");
```

Запрос `Select Distinct`

Другой тип запросов, который узнают те, кто знаком с языком запросов данных SQL — запрос `SELECT DISTINCT`, в котором выполняется поиск внутри данных уникальных, т.е. не повторяющихся, значений. Потребность в этом возникает очень часто при работе с запросами.

Предположим, что нужно извлечь список регионов из данных заказчиков, которые применялись в предыдущих примерах. В используемых данных нет отдельного списка регионов, поэтому необходимо получить список уникальных регионов из списка заказчиков. LINQ предоставляет метод `Distinct()`, который облегчает решение задачи нахождения данных подобного рода. Этот метод используется в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Проекция: запрос `Select Distinct`

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Модифицируйте предыдущий пример, `23-08-ProjectionCreateNewObjects`, или создайте новое консольное приложение по имени `23-09-SelectDistinctQuery` в каталоге `C:\BegVCSharp\Chapter23`.
2. Скопируйте код создания класса `Customer` и инициализации списка `customers` (`List<Customer> customers`) из примера `23-07-QueryComplesObjects`; код остается тем же самым.
3. В методе `Main()` вслед за инициализацией списка заказчиков введите (или модифицируйте) запрос следующим образом:

```
var queryResults = customers.Select(c => c.Region).Distinct();
```

*Фрагмент кода **23-09-SelectDistinctQuery\Program.cs***

4. Остальной код в методе `Main()` оставьте без изменений, как в предыдущем примере.
5. Скомпилируйте и выполните программу. Вы увидите уникальный список регионов, в которых находятся заказчики:

```
North America
Asia
South America
Europe
Africa
Program finished, press Enter/Return to continue:
```

Описание работы

Класс `Customer` и инициализация списка `customers` остаются теми же, что и в предыдущем примере. В операторе запроса вызывается метод `Select()` с простым лямбда-выражением для выбора региона из объектов `Customer`, а затем вызывается `Distinct()`, чтобы вернуть из `Select()` только уникальные результаты:

```
var queryResults = customers.Select(c => c.Region).Distinct();
```

Поскольку `Distinct()` доступен только в синтаксисе методов, применяется вызов `Select()` из синтаксиса методов. Однако вызвать `Distinct()` для модификации запроса можно и в синтаксисе запросов:

```
var queryResults = (from c in customers select c.Region).Distinct();
```

Поскольку синтаксис запросов транслируется компилятором C# в те же последовательности вызовов методов LINQ, как и в синтаксисе методов, синтаксис можно смешивать в случаях, когда это оправдано в плане читабельности и стиля.

Методы `Any` и `All`

Другой тип запроса, который часто может понадобиться, состоит в определении того, есть ли данные, которые удовлетворяют заданному условию, или в определении того, что все данные удовлетворяют заданному условию. Например, может потребоваться определить, что продукта нет на складе (количество равно нулю) или что транзакция произошла.

LINQ предоставляет булевские методы `Any()` и `All()`, которые могут быстро сообщить, истинно ли некоторое условие для существующих данных. Это облегчает задачу поиска данных, которую придется выполнять в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование методов `Any` и `All`

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Модифицируйте предыдущий пример, `23-09-SelectDistinctQuery`, или создайте новое консольное приложение по имени `23-10-AnyAndAll` в каталоге `C:\BegVCSharp\Chapter23`.

2. Скопируйте код создания класса `Customer` и инициализации списка `customers` (`List<Customer> customers`) из примера `23-07-QueryComplexObjects`; код остается тем же самым.

3. В методе `Main()` после инициализации списка `customers` и объявления запроса удалите цикл обработки и введите код, показанный ниже:

```
bool anyUSA = customers.Any(c => c.Country == "USA");
if (anyUSA)
{
  Console.WriteLine("Some customers are in the USA");
                  // Некоторые заказчики находятся в США
}
else
{
  Console.WriteLine("No customers are in the USA");
                  // Нет заказчиков из США
}
bool allAsia = customers.All(c => c.Region == "Asia");
```

```
if (allAsia)
{
  Console.WriteLine("All customers are in Asia");
  // Все заказчики находятся в Азии
}
else
{
  Console.WriteLine("Not all customers are in Asia");
  // Не все заказчики находятся в Азии
}
```

*Фрагмент кода **23-10-AnyAndAll\Program.cs***

4. Остальной код в методе `Main()` такой же, как и в предыдущем примере.
5. Скомпилируйте и выполните программу. Вы увидите сообщения, указывающие на то, что некоторые заказчики находятся в США, но не все — в Азии:

```
Some customers are in the USA
Not all customers are in Asia
Program finished, press Enter/Return to continue:
```

Описание работы

Класс `Customer` и инициализация списка `customers` — такие же, как в предыдущих примерах. В первом операторе запроса вызывается метод `Any()` с простым лямбда-выражением для проверки, имеет ли поле `Country` в `Customer` значение `USA`:

```
bool anyUSA = customers.Any(c => c.Country == "USA");
```

Метод LINQ по имени `Any()` применяет переданное ему лямбда-выражение `c => c.Country == "USA"` ко всем данным в списке `customers` и возвращает `true`, если это лямбда-выражение истинно для любого заказчика в списке.

Затем проверяется результирующая булевская переменная, возвращенная методом `Any()`, и выводится сообщение, отражающее результат запроса (метод `Any()` не просто возвращает `true` или `false` — он выполняет запрос для получения этого результата):

```
if (anyUSA)
{
  Console.WriteLine("Some customers are in the USA");
}
else
{
  Console.WriteLine("No customers are in the USA");
}
```

Хотя можно сделать это сообщение более компактным, применив некоторый искусный код, в приведенном здесь виде оно более наглядно и читабельно. Как и следовало ожидать, переменная `anyUSA` установлена в `true`, т.к. на самом деле в наборе данных присутствуют заказчики, находящиеся в США, поэтому вы видите сообщение `Some customers are in the USA`.

В следующем операторе запроса вызывается метод `All()` с другим простым лямбда-выражением, чтобы определить, все ли заказчики находятся в Азии:

```
bool allAsia = customers.All(c => c.Region == "Asia");
```

Метод LINQ по имени `All()` применяет переданное ему лямбда-выражение к набору данных и возвращает `false`; этого следовало ожидать, потому что некоторые из заказчиков находятся не в Азии. Затем на основе значения `allAsia` выводится соответствующее сообщение.

Многоуровневое упорядочивание

Имея дело с объектами, включающими несколько свойств, вы можете столкнуться с ситуацией, когда упорядочивания результатов запроса по единственному полю оказывается недостаточно. Что, если вы захотите запросить заказчиков и упорядочить результаты в алфавитном порядке по региону, а затем — по стране или названию города внутри региона? LINQ позволяет сделать это очень легко, как вы убедитесь в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Многоуровневое упорядочивание

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Модифицируйте предыдущий пример, `23-08-ProjectionCreateNewObjects`, или создайте новое консольное приложение по имени `23-11-MultiLevelOrdering` в каталоге `C:\BegVCSharp\Chapter23`.
2. Создайте класс `Customer` и инициализацию списка `customers`, как показано в примере `23-07-QueryComplexObjects`; этот код точно такой же, как и в предыдущих примерах.
3. В методе `Main()` после инициализации списка заказчиков введите следующий запрос:

```
var queryResults =
    from c in customers
    orderby c.Region, c.Country, c.City
    select new { c.ID, c.Region, c.Country, c.City }
    ;
```

*Фрагмент кода **23-11-MultiLevelOrdering\Program.cs***

4. Цикл обработки запроса и остальной код метода `Main()` — точно такие же, как и в предыдущих примерах.
5. Скомпилируйте и выполните программу. Вы увидите выбранные свойства из `customers`, упорядоченные по алфавиту сначала по региону, затем по стране и, наконец, по городу, как показано ниже:

```
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = K, Region = North America, Country = Mexico, City = Mexico City }
{ ID = N, Region = North America, Country = USA, City = Los Angeles }
{ ID = A, Region = North America, Country = USA, City = New York }
{ ID = E, Region = South America, Country = Brazil, City = Sao Paulo }
{ ID = S, Region = South America, Country = Colombia, City = Bogota }
{ ID = T, Region = South America, Country = Peru, City = Lima }
Program finished, press Enter/Return to continue:
```

Описание работы

Класс `Customer` и список инициализации `customers` — такие же, как и в предыдущих примерах. В этом запросе нет конструкции `where`, потому что нужно увидеть всех заказчиков, но перечисляются только те поля, которые необходимо отсортировать по порядку, в разделенной запятыми последовательности в конструкции `orderby`:

```
orderby c.Region, c.Country, c.City
```

Проще некуда, не правда ли? Тот факт, что простой список полей разрешен в конструкции `orderby`, но не допускается в конструкции `select`, выглядит несколько не интуитивно понятным, но так уж работает LINQ. Это имеет смысл, если вы поймете, что конструкция `select` создает новый объект, а конструкция `orderby` по определению оперирует полями.

Можно добавить ключевое слово `descending` к любому из перечисленных полей, чтобы изменить порядок сортировки для этого поля. Например, чтобы упорядочить этот запрос по возрастанию региона, но по убыванию страны, просто добавьте в список `descending` после `country`:

```
orderby c.Region, c.Country descending, c.City
```

После этого вы увидите следующее:

```
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = N, Region = North America, Country = USA, City = Los Angeles }
{ ID = A, Region = North America, Country = USA, City = New York }
{ ID = K, Region = North America, Country = Mexico, City = Mexico City }
{ ID = T, Region = South America, Country = Peru, City = Lima }
{ ID = S, Region = South America, Country = Colombia, City = Bogota }
{ ID = E, Region = South America, Country = Brazil, City = Sao Paulo }
Program finished, press Enter/Return to continue:
```

Обратите внимание, что города Индии и Китая по-прежнему расположены в порядке возрастания, в то время как порядок стран изменился на противоположный.

Синтаксис методов многоуровневого упорядочивания: `ThenBy`

Если заглянуть “за кулисы” многоуровневого упорядочивания, использующего синтаксис методов `ThenBy()` и `OrderBy()`, то там все несколько сложнее.

Например, вы получите тот же результат, что и в предыдущем примере, если напишете следующий код:

```
var queryResults = customers.OrderBy(c => c.Region)
                      .ThenBy(c => c.Country)
                      .ThenBy(c => c.City)
                      .Select(c => new { c.ID, c.Region, c.Country, c.City });
```

Теперь более очевидно, почему список из многих полей разрешен в конструкции orderby в синтаксисе запросов; как здесь показано, он транслируется в последовательность вызовов метода ThenBy() для каждого поля. Порядок записи вызовов важен: вы должны начать с OrderBy(), потому что метод ThenBy() доступен только на интерфейсе IOrderedEnumerable, возвращаемом OrderBy(). Однако метод ThenBy() может быть соединен с другим методом ThenBy() столько раз, сколько нужно. Это тот случай, когда синтаксис запросов явно проще в написании, чем синтаксис методов.

Порядок по убыванию указывается вызовом либо OrderByDescending() на первом поле, подлежащем сортировке в убывающем порядке, либо вызовом ThenByDescending(), если любое из остальных полей должно быть отсортировано в убывающем порядке. Чтобы отсортировать страны в порядке убывания, как в этом примере, запрос в синтаксисе методов должен выглядеть следующим образом:

```
var queryResults = customers.OrderBy(c => c.Region)
                        .ThenByDescending(c => c.Country)
                        .ThenBy(c => c.City)
                        .Select(c => new { c.ID, c.Region, c.Country, c.City });
```

Групповые запросы

Групповой запрос делит данные на группы и позволяет сортировать, вычислять агрегатные значения и сравнивать группы. Это часто наиболее интересные запросы в бизнес-контексте (они действительно используются для принятия решений). Например, может понадобиться сравнить продажи по стране и региону, чтобы решить, стоит ли открывать новый магазин или нанимать дополнительный персонал. Давайте сделаем это в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Групповой запрос

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новое консольное приложение по имени 23-12-GroupQuery в каталоге C:\BegVCSharp\Chapter23.
2. Создайте класс Customer и инициализируйте список customers (List<Customer> customers), как показано в примере 23-07-QueryComplexObjects; этот код в точности совпадает с предыдущим примером.
3. В методе Main() вслед за инициализацией списка customers введите два запроса:

```
var queryResults =
    from c in customers
    group c by c.Region into cg
    select new { TotalSales = cg.Sum(c => c.Sales), Region = cg.Key }
    ;
var orderedResults =
    from cg in queryResults
    orderby cg.TotalSales descending
    select cg
    ;
```

*Фрагмент кода **23-12-GroupQuery\Program.cs***

4. Модифицируйте метод Main(), добавив следующий оператор print и цикл обработки foreach:

```
Console.WriteLine("Total\t: By\nSales\t: Region\n-----\t ------");
// Вывод информации о продажах по регионам
foreach (var item in orderedResults)
{
   Console.WriteLine(item.TotalSales + "\t: " + item.Region);
}
```

5. Цикл обработки результатов и остальной код метода `Main()` — такой же, как в предыдущих примерах. Скомпилируйте и выполните программу. Ниже показан групповой результат:

```
Total  : By
Sales  : Region
-----    ------
52997  : Asia
16999  : North America
12444  : Europe
8558   : South America
7000   : Africa
```

Описание работы

Класс `Customer` и инициализация списка `customers` — такие же, как в предыдущих примерах.

Данные в групповом запросе группируются по ключевому полю — полю, для которого все члены каждой группы разделяют общее значение. В этом примере ключевым полем является `Region`:

```
group c by c.Region
```

Для каждой группы необходимо вычислить сумму, поэтому выполняется группирование в новый результирующий набор по имени `cg`:

```
group c by c.Region into cg
```

В конструкции `select` проектируется новый анонимный тип, свойствами которого будут общая сумма продаж (вычисляемая по обращению к результирующему набору `cg`) и ключевое значение группы, к которому вы обращаетесь по специальному полю `Key` группы:

```
select new { TotalSales = cg.Sum(c => c.Sales), Region = cg.Key }
```

Групповой результирующий набор реализует интерфейс LINQ по имени `IGrouping`, который поддерживает свойство `Key`. При обработке групповых результатов почти всегда нужно каким-то образом обращаться к свойству `Key`, т.к. оно представляет критерий создания каждой группы данных.

Результат должен быть упорядочен по убыванию значений в поле `TotalSales`, что позволит увидеть, какой регион имеет максимальный уровень общих продаж, какой регион занимает по этому показателю второе место, третье и т.д. Для этого создается второй запрос, упорядочивающий результаты группового запроса:

```
var orderedResults =
    from cg in queryResults
    orderby cg.TotalSales descending
    select cg
    ;
```

Второй запрос — это стандартный запрос `select` с конструкцией `orderby`, который демонстрировался в предыдущих примерах; он не использует никаких групповых средств LINQ, за исключением того, что источником данных для него является предыдущий групповой запрос.

Затем выводятся результаты с небольшой долей форматирующего кода для отображения данных с заголовками столбцов и некоторыми разделителями между суммами и именами групп:

```
Console.WriteLine("Total\t: By\nSales\t: Region\n-----\t ------");
foreach (var item in orderedResults)
{
   Console.WriteLine(item.TotalSales + "\t: " + item.Region);
};
```

Вывод может быть сформатирован более изощренным образом с указанием ширин полей и способа выравнивания итоговых сумм, но это только пример, потому заботиться об этом не обязательно — вы и так можете достаточно ясно увидеть результирующие данные и понять, что делает код.

Методы `Take` и `Skip`

Предположим, что в существующем наборе данных нужно найти первые пять заказчиков по уровню продаж. Вы не знаете заранее, какой объем продаж позволит включить заказчика в группу из пяти "передовиков", поэтому вы не можете использовать условие `where` для их поиска.

Некоторые базы данных SQL, такие как Microsoft SQL Server, реализуют операцию `TOP`, так что можно выдать команду вроде `SELECT TOP 5 FROM...` и получить первые пять заказчиков.

LINQ-эквивалент этой операции представлен методом `Take()`, который принимает первые *n* результатов в выводе запроса. При практическом применении это должно быть скомбинировано с `orderby` для получения первых *n* результатов. Однако применять `orderby` не обязательно, поскольку могут быть ситуации, когда заранее известно, что данные уже находятся в нужном порядке, или же когда по каким-то причинам необходимо получить первые *n* результатов, не заботясь об их порядке.

Противоположностью `Take()` является метод `Skip()`, который пропускает первые *n* результатов, возвращая остальные. `Take()` и `Skip()` называют в документации LINQ *разделяющими операциями* (partitioning operators), потому что они разбивают результирующий набор на первые *n* результатов (`Take()`) и/или остальные (`Skip()`).

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование методов `Take` и `Skip`

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новое консольное приложение по имени `23-13-TakeAndSkip` в каталоге `C:\BegVCSharp\Chapter23`.
2. Скопируйте код создания класса `Customer` и инициализации списка `customers` (`List<Customer> customers`) из примера `23-07-QueryComplexObjects`.
3. В методе `Main()` вслед за инициализацией списка `customers` введите следующий запрос:

```
// Синтаксис запроса
var queryResults =
    from c in customers
    orderby c.Sales descending
    select new { c.ID, c.City, c.Country, c.Sales }
    ;
```

*Фрагмент кода **23-13-TakeAndSkip\Program.cs***

4. Добавьте два цикла обработки результатов, один с использованием `Take()`, а другой — со `Skip()`:

```
Console.WriteLine("Top Five Customers by Sales");
// Верхняя пятерка заказчиков по результатам продаж
foreach (var item in queryResults.Take(5))
{
   Console.WriteLine(item);
}
Console.WriteLine("Customers Not In Top Five");
// Заказчики, не вошедшие в верхнюю пятерку
foreach (var item in queryResults.Skip(5))
{
   Console.WriteLine(item);
}
```

5. Скомпилируйте и выполните программу. Вы увидите пять ведущих заказчиков и затем остальных заказчиков, как показано ниже:

```
Top Five Customers by Sales
{ ID = A, City = New York, Country = USA, Sales = 9999 }
{ ID = R, City = Beijing, Country = China, Sales = 9000 }
{ ID = B, City = Mumbai, Country = India, Sales = 8888 }
{ ID = Q, City = London, Country = UK, Sales = 8000 }
{ ID = C, City = Karachi, Country = Pakistan, Sales = 7777 }
Customers Not In Top Five
{ ID = P, City = Tehran, Country = Iran, Sales = 7000 }
{ ID = D, City = Delhi, Country = India, Sales = 6666 }
{ ID = O, City = Cairo, Country = Egypt, Sales = 6000 }
{ ID = E, City = Sao Paulo, Country = Brazil, Sales = 5555 }
{ ID = N, City = Los Angeles, Country = USA, Sales = 5000 }
{ ID = F, City = Moscow, Country = Russia, Sales = 4444 }
{ ID = M, City = Tokyo, Country = Japan, Sales = 4000 }
{ ID = G, City = Seoul, Country = Korea, Sales = 3333 }
{ ID = L, City = Jakarta, Country = Indonesia, Sales = 3000 }
{ ID = H, City = Istanbul, Country = Turkey, Sales = 2222 }
{ ID = T, City = Lima, Country = Peru, Sales = 2002 }
{ ID = K, City = Mexico City, Country = Mexico, Sales = 2000 }
{ ID = I, City = Shanghai, Country = China, Sales = 1111 }
{ ID = S, City = Bogot б , Country = Colombia, Sales = 1001 }
{ ID = J, City = Lagos, Country = Nigeria, Sales = 1000 }
Program finished, press Enter/Return to continue:
```

Описание работы

Класс `Customer` и инициализация списка `customers` — такие же, как в предыдущих примерах.

Основной запрос состоит из конструкции `from...orderby...select` в синтаксисе запросов. Он похож на ранее создаваемые в главе запросы, но с тем отличием, что здесь отсутствует ограничивающая конструкция `where`, потому что необходимо получить всех заказчиков (упорядоченных по объемам продаж от большего к меньшему):

```
var queryResults =
   from c in customers
   orderby c.Sales descending
   select new { c.ID, c.City, c.Country, c.Sales }
```

Этот пример работает несколько иначе, чем предыдущие, в том смысле, что операция не применяется до тех пор, пока не начнется цикл `foreach` на его результатах, поскольку эти результаты должны использоваться повторно. Сначала с помощью `Take(5)` получаются первые пять заказчиков:

```
foreach (var item in queryResults.Take(5))
```

Затем посредством `Skip(5)` пропускаются первые пять элементов (которые уже выведены на консоль) и выводятся остальные заказчики из того же самого результирующего набора:

```
foreach (var item in queryResults.Skip(5))
```

Код для вывода результатов и приостановки вывода на экран такой же, как в предыдущих примерах, за исключением небольших изменений в сообщениях, поэтому здесь он не объясняется.

Методы `First` и `FirstOrDefault`

Предположим, что необходимо найти пример заказчика из Африки в наборе данных. Нужны сами данные, а не значение `true/false` или результирующий набор соответствующих значений.

LINQ предоставляет эту возможность через метод `First()`, который возвращает первый элемент результирующего набора, отвечающий заданным критериям. Если нет ни одного заказчика из Африки, то LINQ предоставляет метод для обработки такого случая без дополнительного кода обработки ошибок: `FirstOrDefault()`.

В следующем практическом занятии используется как `First()`, так и `FirstOrDefault()` со списком данных о заказчиках.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование методов `First` и `FirstOrDefault`

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новое консольное приложение по имени `23-14-FirstOrDefault` в каталоге `C:\BegVCSharp\Chapter23`.
2. Скопируйте код создания класса `Customer` и инициализации списка `customers` (`List<Customer> customers`) из примера `23-07-QueryComplexObjects`.
3. В методе `Main()` вслед за инициализацией списка заказчиков введите следующий запрос:

```
var queryResults = from c in customers
                   select new { c.City, c.Country, c.Region }
                   ;
```

*Фрагмент кода **23-14-FirstOrDefault\Program.cs***

4. Введите следующие запросы, в которых используются методы `First()` и `FirstOrDefault()`:

```
Console.WriteLine("A customer in Africa");
                // Заказчик из Африки
Console.WriteLine(queryResults.First(c => c.Region == "Africa"));
Console.WriteLine("A customer in Antarctica");
                // Заказчик из Антарктиды
Console.WriteLine(queryResults.FirstOrDefault(c => c.Region == "Antarctica"));
```

5. Скомпилируйте и выполните программу. Ниже показан результирующий вывод:

```
A customer in Africa
{ City = Lagos, Country = Nigeria, Region = Africa }
A customer in Antarctica
Program finished, press Enter/Return to continue:
```

Описание работы

Класс `Customer` и инициализация списка `customers` — такие же, как в предыдущих примерах.

Основной запрос состоит из конструкции `from...orderby...select` в синтаксисе запросов, как и те, что создавались ранее в этой главе, но без частей `where` и `orderby`. С помощью `select` проектируются интересующие поля, в рассматриваемом случае это свойства `City`, `Country` и `Region`:

```
var queryResults = from c in customers
    select new { c.City, c.Country, c.Region }
    ;
```

Поскольку операция `First()` возвращает значение единственного объекта, а не результирующий набор, создавать цикл `foreach` не нужно; вместо этого результат выводится непосредственно:

```
Console.WriteLine(queryResults.First(c => c.Region == "Africa"));
```

Этот запрос находит заказчика, и результат `City = Lagos, Country = Nigeria, Region = Africa` выводится на консоль. Затем осуществляется запрос заказчика из Антарктиды с использованием `FirstOrDefault()`:

```
Console.WriteLine(queryResults.FirstOrDefault(c => c.Region == "Antarctica"));
```

Этот запрос ничего не находит, поэтому возвращается `null` (пустой результат), который выглядит как пустая строка. Что случилось бы, если в таком запросе применить операцию `First()` вместо `FirstOrDefault()`? В этом случае было бы сгенерировано следующее исключение:

```
System.InvalidOperationException: Sequence contains no matching element
System.InvalidOperationException: Последовательность не содержит совпадающего элемента
```

Вместо этого `FirstOrDefault()` возвращает элемент по умолчанию для списка, если критерий поиска не удовлетворяется ни одной записью, и этим элементом по умолчанию является `null` для данного анонимного типа. Иначе для региона Антарктиды возникло бы исключение.

Код вывода результатов на консоль и приостановки вывода — такой же, как и в предыдущих примерах, за исключением небольших изменений в экранных сообщениях.

Операции с множествами

LINQ поддерживает стандартные операции работы с множествами, такие как `Union()` и `Intersect()`, которые оперируют с результатами запроса. Вы использовали одну из этих операций, когда ранее писали запрос `Distinct()`.

В следующем практическом занятии вы добавите простой список заказов, который прислан гипотетическим заказчиком, и воспользуетесь стандартными операциями с множествами для сопоставления заказов с существующими заказчиками.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование операций с множествами

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новое консольное приложение по имени `23-15-SetOperators` в каталоге `C:\BegVCSharp\Chapter23`.
2. Скопируйте код создания класса `Customer` и инициализации списка `customers` (`List<Customer> customers`) из примера `23-07-QueryComplexObjects`.

3. После объявления класса `Customer` добавьте следующий класс `Order`:

```
class Order
{
   public string ID { get; set; }
   public decimal Amount { get; set; }
}
```

*Фрагмент кода **23-15-SetOPerators\Program.cs***

4. В методе `Main()`, вслед за инициализацией списка `customers`, создайте и инициализируйте список `orders` данными, показанными ниже:

```
List<Order> orders = new List<Order> {
  new Order { ID="P", Amount=100 },
  new Order { ID="Q", Amount=200 },
  new Order { ID="R", Amount=300 },
  new Order { ID="S", Amount=400 },
  new Order { ID="T", Amount=500 },
  new Order { ID="U", Amount=600 },
  new Order { ID="V", Amount=700 },
  new Order { ID="W", Amount=800 },
  new Order { ID="X", Amount=900 },
  new Order { ID="Y", Amount=1000 },
  new Order { ID="Z", Amount=1100 }
};
```

5. После инициализации списка `orders` добавьте следующие запросы:

```
var customerIDs =
   from c in customers
   select c.ID
   ;
var orderIDs =
   from o in orders
   select o.ID
   ;
```

6. Введите следующий запрос, в котором используется метод `Intersect()`:

```
var customersWithOrders = customerIDs.Intersect(orderIDs);
Console.WriteLine("Customers IDs with Orders:");
                  // Идентификаторы заказчиков с заказами
foreach (var item in customersWithOrders)
{
   Console.Write("{0} ", item);
}
Console.WriteLine();
```

7. Затем добавьте запрос, использующий метод `Except()`:

```
Console.WriteLine("Order IDs with no customers:");
                // Идентификаторы заказов без заказчиков
var ordersNoCustomers = orderIDs.Except(customerIDs);
foreach (var item in ordersNoCustomers)
{
   Console.Write("{0} ", item);
}
Console.WriteLine();
```

8. И, наконец, введите следующий запрос, в котором используется метод `Union()`:

```
Console.WriteLine("All Customer and Order IDs:");
                // Все идентификаторы заказчиков и заказов
var allCustomerOrderIDs = orderIDs.Union(customerIDs);
```

```
foreach (var item in allCustomerOrderIDs)
{
   Console.Write("{0} ", item);
}
Console.WriteLine();
```

9. Скомпилируйте и выполните программу. В результате получится такой вывод:

```
Customers IDs with Orders:
P Q R S T
Order IDs with no customers:
U V W X Y Z
All Customer and Order IDs:
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Program finished, press Enter/Return to continue:
```

Описание работы

Класс `Customer` и инициализация списка `customers` — те же, что и в предыдущих примерах. Новый класс `Order` подобен классу `Customer` и использует средство автоматических свойств C# для объявления общедоступных свойств (`ID`, `Amount`):

```
class Order
{
   public string ID { get; set; }
   public decimal Amount { get; set; }
}
```

Как и класс `Customer`, это является упрощенным примером с количеством данных, достаточным для демонстрации выполнения запроса.

Для получения полей `ID` из классов `Customer` и `Order` применяются два простых запроса `from...select`:

```
var customerIDs =
    from c in customers
    select c.ID
    ;
var orderIDs =
    from o in orders
    select o.ID
    ;
```

Затем с помощью операции множества `Intersect()` находятся идентификаторы только тех заказчиков, которые имеют соответствующие заказы в результирующем наборе `orderIDs`. В пересечение включаются только те идентификаторы, которые присутствуют в обоих результирующих наборах:

```
var customersWithOrders = customerIDs.Intersect(orderIDs);
```

Операции с множествами требуют, чтобы члены множеств имели одинаковый тип, для гарантии ожидаемого результата. Здесь используется преимущество того факта, что идентификаторы в обоих типах объектов являются строками и обладают одинаковой семантикой (подобно внешним ключам в базе данных).

При выводе результатов учитывается тот факт, что идентификаторы представлены одиночным символом, поэтому применяется вызов `Console.Write()` без `Console.WriteLine()` до конца цикла `foreach`, чтобы сделать вывод компактным и четким:

```
Console.WriteLine("Customers IDs with Orders:");
foreach (var item in customersWithOrders)
{
   Console.Write("{0} ", item);
}
Console.WriteLine();
```

Та же логика вывода на консоль используется и в остальных циклах foreach.

Затем с помощью операции Except() находятся идентификаторы заказов, которые не имеют соответствующего заказчика:

```
Console.WriteLine("Order IDs with no customers:");
var ordersNoCustomers = orderIDs.Except(customerIDs);
```

И, наконец, посредством операции Union() получается объединение всех полей идентификаторов заказчиков с полями идентификаторов заказов:

```
Console.WriteLine("All Customer and Order IDs:");
var allCustomerOrderIDs = orderIDs.Union(customerIDs);
```

Обратите внимание, что идентификаторы выводятся в том же порядке, в каком они появляются в списках заказчиков и заказов, с исключением дубликатов.

Код приостановки вывода на экран такой же, как и в предыдущих примерах.

Операции с множествами удобны, хотя практическая польза от их применения ограничена требованием, чтобы все объекты, которыми они манипулируют, относились к одному и тому же типу. Эти операции удобны в определенных ситуациях, когда нужно манипулировать наборами одинаково типизированных результатов, но в наиболее типичном случае, когда требуется работа с различными связанными типами объектов, понадобится более практичный механизм работы с разными типами, такой как оператор соединения join.

Соединения

Наборы данных, подобные спискам customers и orders, которые были только что созданы с общим ключевым полем (ID), позволяют использовать запрос join, с помощью которого можно запросить взаимосвязанные данные из обоих списков в единственном запросе, объединяя их по значению ключевого поля. Это подобно оператору JOIN в языке SQL. Как и можно было ожидать, LINQ предоставляет команду join в синтаксисе запросов, которой вы воспользуетесь в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Получение соединения

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новое консольное приложение по имени 23-16-JoinQuery в каталоге C:\BegVCSharp\Chapter23.
2. Скопируйте создание классов Customer и Order, а также инициализации списков customers (List<Cistomer> customers) и orders (List<Order> orders) из предыдущего примера; этот код здесь точно такой же.
3. В методе Main(), вслед за инициализацией списков customers и orders, введите следующий запрос:

```
var queryResults = from c in customers
    join o in orders on c.ID equals o.ID
    select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder = o.Amount,
                         SalesAfter = c.Sales+o.Amount };
```

*Фрагмент кода **23-16-JoinQuery\Program.cs***

4. Завершите программу стандартным циклом обработки запроса foreach, который применялся в предыдущих примерах:

```
foreach (var item in queryResults)
{
   Console.WriteLine(item);
}
```

5. Скомпилируйте и выполните программу. В результате будет получен следующий вывод:

```
{ ID = P, City = Tehran, SalesBefore = 7000, NewOrder = 100, SalesAfter = 7100 }
{ ID = Q, City = London, SalesBefore = 8000, NewOrder = 200, SalesAfter = 8200 }
{ ID = R, City = Beijing, SalesBefore = 9000, NewOrder = 300, SalesAfter = 9300 }
{ ID = S, City = Bogot б , SalesBefore = 1001, NewOrder = 400, SalesAfter = 1401 }
{ ID = T, City = Lima, SalesBefore = 2002, NewOrder = 500, SalesAfter = 2502 }
Program finished, press Enter/Return to continue:
```

Описание работы

Код объявления и инициализации класса `Customer` и класса `Order`, а также списков `customers` и `orders` — такой же, как и в предыдущем примере.

В запросе ключевое слово `join` служит для соединения каждого заказчика с соответствующими ему заказами, используя поле `ID` из классов `Customer` и `Order`, соответственно:

```
var queryResults =
    from c in customers
    join o in orders on c.ID equals o.ID
```

Ключевое слово `on` следует за именем поля ключа (`ID`), а ключевое слово `equals` указывает на соответствующее поле в другой коллекции. Результат запроса включает только данные объектов, имеющих то же значение поля `ID`, что и соответствующее поле `ID` в другой коллекции.

Оператор `select` проектирует новый тип данных со свойствами, названными так, что вы можете ясно видеть исходную сумму продаж, новый заказ и результирующую новую сумму:

```
select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder = o.Amount,
                           SalesAfter = c.Sales+o.Amount };
```

Хотя в этом примере объем продаж в объекте `customer` не увеличивается, сделать это несложно.

Логика цикла `foreach` и вывод значений, возвращенных запросом, в точности такие же, как и в предыдущих программах настоящей главы.

Резюме

Как вы убедились, LINQ делает запросы, написанные на естественном языке C#, довольно легкими и мощными. В следующей главе будет показано, как использовать LINQ to SQL для запросов реляционных баз данных и эффективной работы с крупными наборами данных.

В синтаксисе методов LINQ доступно слишком много методов, чтобы охватить их все в книге для начинающих. За дополнительными деталями и примерами обращайтесь к онлайновой документации по LINQ от Microsoft. Короткие примеры каждого из методов LINQ вы найдете в разделе “101 LINQ Samples” в справочной системе MSDN (или по адресу `http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx`).

Хорошим источником для изучения функционального программирования в контексте LINQ может послужить руководство Эрика Вайта (Eric White) по функциональному программированию, доступное по адресу `http://blogs.msdn.com/b/ericwhite/`. Кроме того, там предлагается сжатое руководство по синтаксису методов LINQ.

Упражнения

1. Модифицируйте программу из первого примера (`23-01-FirstLINQquery`) для упорядочивания результатов по убыванию.

2. Измените число, переданное методу `generateLotsOfNumbers()` в примере с большими числами (`23-05-LargeNumberQuery`), для создания результирующих наборов разного размера и посмотрите, как это повлияет на результаты запроса.
3. Добавьте конструкцию `orderby` к запросу в примере с большими числами (`23-05-LargeNumberQuery`), чтобы посмотреть, как это влияет на производительность.
4. Модифицируйте условия запроса в примере программы с большими числами (`23-05-LargeNumberQuery`) для выбора больших и меньших подмножеств списка чисел. Как это влияет на производительность?
5. Измените пример синтаксиса методов (`23-02-LINQMethodSyntax`), чтобы полностью исключить конструкцию `where`. Насколько объемный вывод он сгенерирует?
6. Модифицируйте пример программы запроса сложных объектов (`23-07-QueryComplexObjects`) для выбора другого подмножества полей запроса с условием, соответствующим этим полям.
7. Добавьте агрегатные операции к программе из первого примера (`23-01-FirstLINQquery`). Какие простые агрегатные операции доступны для нечисловых результирующих наборов?

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Что такое LINQ и когда этот язык используется	LINQ — это язык запросов, встроенный в C#. Он используется для отправки запросов к большим коллекциям объектов, XML или базам данных.
Части запроса LINQ	Запрос LINQ включает конструкции `from`, `where`, `select` и `orderby`.
Как получить результаты запроса LINQ	Для прохода в цикле по результатам запроса LINQ применяется оператор `foreach`.
Отложенное выполнение	Выполнение запроса LINQ откладывается до начала выполнения оператора `foreach`.
Синтаксис методов и синтаксис запросов	Синтаксис запросов используется для простых запросов LINQ, а синтаксис методов — для более сложных запросов. Для любого запроса оба синтаксиса дают один и тот же результат.
Агрегатные операции	Агрегатные операции LINQ позволяют получить информацию о крупном наборе данных без необходимости проходить по нему в цикле.
Проекция	Проекция служит для изменения типов данных и создания новых объектов в запросах.
Групповые запросы	Групповые запросы используются для разделения данных по группам с последующей сортировкой, вычислением результатов агрегатных операций и сравнения по группам.
Упорядочение	Операция `orderby` позволяет упорядочивать результаты запроса.
Операции с множествами	Операции с множествами `Union()`, `Intersect()` и `Distinct()` служат для поиска совпадающих данных во множественных результирующих наборах.
Соединения	Операция `join` позволяет запрашивать связанные данные в нескольких коллекциях с помощью одного запроса.

24

Применение LINQ

В ЭТОЙ ГЛАВЕ...

- Вариации LINQ
- Использование LINQ с базами данных
- Навигация по отношениям в базе данных
- Использование LINQ с XML
- Использование конструкторов LINQ to XML
- Генерация документов XML из баз данных
- Работа с фрагментами XML

В предыдущей главе было дано введение в LINQ (Language Integrated Query — язык интегрированных запросов) и показано, как LINQ работает с объектами. В этой главе вы научитесь применять LINQ к запросам и манипулировать данными из разных источников, таких как базы данных и документы XML.

Вариации LINQ

Среда Visual Studio 2010 и платформа .NET Framework 4 поставляются с множеством встроенных возможностей LINQ, которые предлагают решения для запросов разнообразных типов данных.

- **LINQ to Objects.** Предоставляет запросы для любой разновидности объектов C#, находящихся в памяти, таких как массивы, списки и другие типы коллекций.
- **LINQ to XML.** Обеспечивает создание и управление документами XML с использованием того же синтаксиса и общего механизма запросов, что и другие вариации LINQ.
- **LINQ to ADO.NET.** ADO.NET, или Active Data Objects (Активные объекты данных) для .NET — общий термин, который охватывает различные классы и библиотеки .NET, предназначенные для доступа к базам данных, в том числе Microsoft SQL Server, Oracle и т.п. LINQ to ADO.NET включает в себя LINQ to Entities, LINQ to DataSet и LINQ to SQL.
- **LINQ to Entities.** ADO.NET Entity Framework — новейший набор классов интерфейса к данным в .NET 4, рекомендованный Microsoft для современной разработки. В этой главе вы добавите источник данных ADO.NET Entity Framework к проекту Visual C#, после чего будете запрашивать его, используя LINQ to Entities.
- **LINQ to DataSet.** Объект `DataSet` появился в первой версии .NET Framework. Вариация LINQ to DataSet позволяет отправлять запросы к унаследованным источникам данных .NET с помощью LINQ.
- **LINQ to SQL.** Альтернативный интерфейс LINQ для .NET 3.5, ориентированный главным образом на Microsoft SQL Server, который в версии .NET 4 был заменен LINQ to Entities.
- **PLINQ.** PLINQ, или Parallel LINQ (Параллельный LINQ), расширяет LINQ to Objects библиотекой для параллельного программирования, которая позволяет распараллеливать выполнение запроса на многоядерном процессоре.

При таком богатстве вариаций LINQ невозможно охватить их все в книге для начинающих, поэтому в настоящей главе будет показано только, как применять LINQ к наиболее распространенным источникам данных — сущностям XML и реляционных баз данных. LINQ работает очень похоже для различных источников данных, так что если вы научитесь пользоваться двумя или тремя вариациями LINQ, то легко сможете применять LINQ также и к новым источникам данных.

Использование LINQ с базами данных

Базы данных SQL, такие как Microsoft SQL Server и Oracle, называются *реляционными базами данных*. Реляционные базы данных построены на основе модели *сущность-отношение* (entity-relationship), где под сущностью понимается абстрактная концепция такого объекта данных, как, например, заказчик, который связан с другими сущностями, подобными заказам и товарам (скажем, заказчик размещает заказ на товары).

В реляционных базах данных используется язык SQL (*Structured Query Language* — язык структурированных запросов) для опроса и манипулирования данными. Традиционно

работа с такой базой данных требует, как минимум, некоторых знаний SQL и затем либо встраивания операторов SQL в язык программирования, либо передачи строк, содержащих операторы SQL, API-вызовам или методам библиотеки классов SQL-ориентированной базы данных.

Звучит сложно, не правда ли? Однако хорошая новость состоит в том, что Visual Studio 2010 и ADO.NET Entity Framewrok могут создавать объекты C#, представляющие сущности модели базы данных, после чего брать на себя все детали взаимодействия с базой данных SQL! Они автоматически транслируют запросы LINQ в операторы SQL и позволяют программам просто взаимодействовать с объектами C#.

Разработка кода для создания набора классов и коллекций, соответствующих существующей структуре реляционной таблицы, представляет собой утомительную и трудоемкую задачу. Но благодаря объектно-реляционному отображению LINQ to Entities, классы, которые соответствуют таблице, создаются автоматически из самой базы данных, так что этого не придется делать вручную, и можно немедленно приступать к использованию этих классов.

Установка программного обеспечения SQL Server и базы данных примеров Northwind

Для запуска примеров, рассмотренных в этой главе, понадобится установить ПО Microsoft Server Express — облегченную версию Microsoft SQL Server.

Если вы знакомы с SQL Server и имеете доступ к экземпляру Microsoft SQL Server 2005 Standard Edition или Microsoft SQL Server 2005 Enterprise Edition с установленной базой данных примеров Northwind, можно пропустить описанные ниже действия по установке. Правда, в этом случае придется изменить информацию подключения для соответствия конкретному экземпляру SQL Server. Если вы ранее не работали с SQL Server, то установите программное обеспечение SQL Server Express.

Установка SQL Server Express 2008

Как Visual Studio 2010, так и Visual C# Express Edition включают копию SQL Server Express — облегченную настольную версию серверного ПО SQL Server 2008.

Если вы уже установили Visual Studio 2010 или Visual C# 2010 Express, но не установили SQL Server 2008 Express Edition, то можете загрузить и установить это программное обеспечение, пройдя по следующему адресу: `http://www.microsoft.com/express/sql/default.aspx`.

Установка базы данных примеров Northwind

База данных Northwind для SQL Server необходима для работы с примерами настоящей главы. Она не входит в состав Visual C# 2010 или SQL Server 2008 Express, но доступна в виде отдельной загрузки на сайте Microsoft. Можете найти с помощью поисковой строки "northwind sample database download" в Google или аналогичном поисковом сайте либо просто перейти по следующему URL:

```
http://www.microsoft.com/downloads/details.
  aspx?FamilyID=06616212-0356-46a0-8da2-eebc53a68034&displaylang=en
```

По этой ссылке загрузится установочный файл `SQL2000SampleDb.msi`.

Запуск файла `.msi` приводит к установке файлов базы данных примеров Northwind. Примите опции по умолчанию на всех экранах мастера установки. По завершении файлы баз данных будут установлены в `C:\SQL Server 2000 Sample Databases\NORTHWND.MDF`.

Имя MDF-файла базы Northwind выглядит как `NORTHWND.MDF` *(без "I").*

Запомните это имя файла и путь, т.к. к ним придется обращаться позднее, при установке соединения с базой данных. На этом установка SQL Express и данных примеров, необходимых в этой главе, завершена. Теперь можно приступать к исследованиям LINQ to Entities.

Первый запрос LINQ к базе данных

В следующем практическом занятии мы создадим простой запрос для поиска подмножества объектов заказчиков в базе данных примеров SQL Server с использованием LINQ to SQL и выведем их на консоль.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Построение первого запроса LINQ к базе данных

Выполните следующие шаги для создания примера в Visual C# 2010.

1. Создайте новый проект консольного приложения по имени `BegVCSharp_24_1_FirstLINQtoDatabaseQuery` в каталоге `C:\BegVCSharp\Chapter24`.
2. Щелкните на кнопке OK для создания проекта.
3. Для добавления источника данных LINQ to Entities для базы данных Northwind перейдите в окно Solution Explorer, щелкните правой кнопкой на проекте `BegVCSharp_24_1_FirstLINQtoDataQuery` и выберите в контекстном меню пункт Data⇨Add New Data Source... (Данные⇨Добавить новый источник данных).
4. В диалоговом окне Choose a Data Source Type (Выбор типа источника данных) выберите Database (База данных).
5. В диалоговом окне Choose a Database Model (Выбор модели базы данных) выберите Entity Data Model (Сущностная модель данных).
6. В диалоговом окне Choose Model Contents (Выбор содержимого модели) выберите Generate From Database (Генерировать из базы данных).
7. В диалоговом окне Choose Your Data Connection (Выбор соединения с базой данных) выберите New Connection (Новое соединение).
8. В диалоговом окне Connection Properties (Свойства соединения) щелкните на кнопке Browse (Обзор) справа от текстового поля Database File Name (new or existing) (Имя файла базы данных (новой или существующей)). В открывшемся диалоговом окне перейдите в каталог `C:\SQL Server 2000 Sample Databases\`, куда была установлена база данных Northwind, и выберите файл `NORTHWND.MDF`. Щелкните на кнопке OK для закрытия диалоговых окон и завершения импорта сущностной модели.
9. В диалоговом окне Choose Your Database Objects (Выбор объектов базы данных) раскройте элемент управления Tables (Таблицы) и отметьте флажки возле элементов `Customers`, `Orders` и `Order Details`. Щелкните на кнопке Finish (Готово) и вы увидите диаграмму сущностных объектов данных в окне, помеченном `Model1.edmx`, как показано на рис. 24.1.
10. Скомпилируйте проект, чтобы объект `Customer` стал доступным, когда вы начнете вводить код на следующем шаге.

Чтобы просмотреть код классов, сгенерированный для сущностной модели, загляните в файл `Model1.designer.cs`, который появляется под исходным файлом `Model1.edmx` в окне Solution Explorer, подобно тому, как сгенерированный код формы помещается в *<имя_формы>*`.designer.cs`. Однако, как и в случае сгенерированного кода формы, модифицировать сгенерированный визуальным конструктором код нельзя, поэтому лучше не открывать этот код в редакторе, за исключением случая, когда необходимо сверить имя класса или уточнить сгенерированный тип данных.

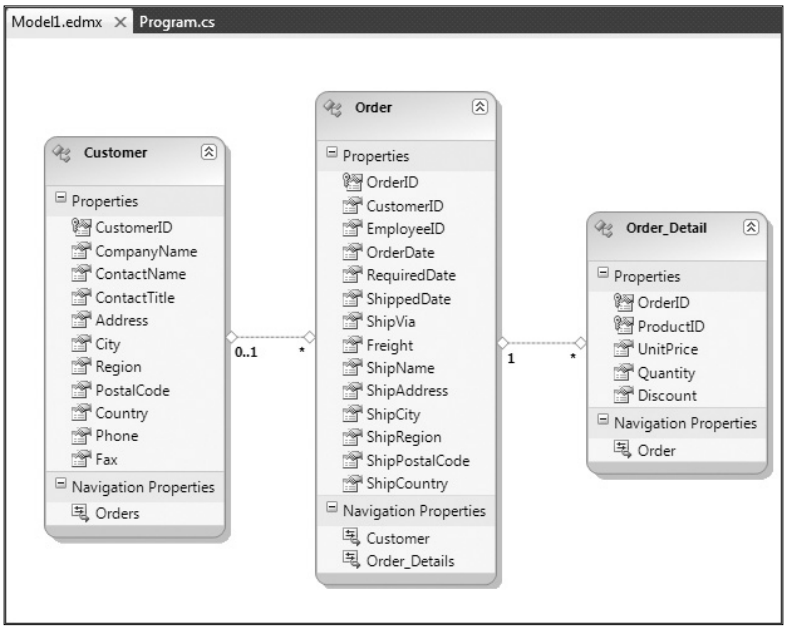


Рис. 24.1. *Диаграмма сущностных объектов данных*

11. Откройте файл `Program.cs` и добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
   NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
   var queryResults = from c in northWindEntities.Customers
                      where c.Country == "USA"
                      select new {
                               ID=c.CustomerID,
                               Name=c.CompanyName,
                               City=c.City,
                               State=c.Region
                             };
      foreach (var item in queryResults) {
         Console.WriteLine(item);
      };
      Console.WriteLine("Press Enter/Return to continue..");
      Console.ReadLine();
}
```

Фрагмент кода ***BegVCSharp\Chapter24\BegVCSharp24_1_FirstLINQtoDatabaseQuery\Program.cs***

12. Скомпилируйте и выполните программу (можно просто нажать <F5> для запуска отладки). Вы увидите информацию о заказчиках из США в таком виде, как показано ниже:

```
{ ID = GREAL, Name = Great Lakes Food Market, City = Eugene, State = OR }
{ ID = HUNGC, Name = Hungry Coyote Import Store, City = Elgin, State = OR }
{ ID = LAZYK, Name = Lazy K Kountry Store, City = Walla Walla, State = WA }
{ ID = LETSS, Name = Let's Stop N Shop, City = San Francisco, State = CA }
{ ID = LONEP, Name = Lonesome Pine Restaurant, Ci.ty = Portland, State = OR }
{ ID = OLDWO, Name = Old World Delicatessen, City = Anchorage, State = AK }
{ ID = RATTC, Name = Rattlesnake Canyon Grocery, City = Albuquerque, State = NM }
{ ID = SAVEA, Name = Save-a-lot Markets, City = Boise, State = ID }
{ ID = SPLIR, Name = Split Rail Beer & Ale, City = Lander, State = WY }
{ ID = THEBI, Name = The Big Cheese, City = Portland, State = OR }
{ ID = THECR, Name = The Cracker Box, City = Butte, State = MT }
{ ID = TRAIH, Name = Trail's Head Gourmet Provisioners, City = Kirkland, State = WA }
{ ID = WHITC, Name = White Clover Markets, City = Seattle, State = WA }
Press Enter/Return to continue...
```

Нажмите клавишу <Enter> для завершения программы и закрытия экрана консоли. Если для запуска использовалась комбинация клавиш <Ctrl+F5> (запуск без отладки), возможно, <Enter> понадобится нажать два раза. Это завершит работу программы. Теперь давайте детально разберем, как она работает.

Описание работы

Код для этого и всех прочих примеров этой главы подобен примерам, описанным в предыдущей главе. В нем используются классы расширения из пространства имен `System.Linq`, на которое ссылается оператор `using`, автоматически добавленный Visual C# 2010 при создании проекта:

```
using System.Linq;
```

Первый шаг в использовании классов LINQ to Entities состоит в создании экземпляра `ObjectContext` для определенной базы данных, к которой вы обращаетесь — класса, который компилируется из файла `.edmx`, созданного в источнике данных. Этот объект является шлюзом к базе данных и предоставляет методы, которые нужны для управления ею из программы. Он также служит фабрикой для создания бизнес-объектов, соответствующих концептуальным сущностям, которые хранятся в базе данных (например, заказчикам и товарам).

В рассматриваемом проекте класс контекста данных называется `NORTHWNDEntities` и компилируется из файла `Model1.edmx`. На первом шаге в методе `Main()` создается экземпляр `NORTHWNDEntities`, как показано ниже:

```
NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
```

Когда вы отмечаете таблицу `Customers` в диалоговом окне Choose Your Database Objects, к классу LINQ to Entities в `Model1.edmx` добавляется объект `Customer`, а к объекту `northWindDataEntities` — член `Customer`, чтобы позволить запрашивать объекты `Customer` в базе данных Northwind.

Оператор LINQ выполняет запрос, используя член `Customer` объекта `northWind Entities` в качестве источника данных:

```
var queryResults = from c in northWindEntities.Customers
                   where c.Country == "USA"
                   select new {
                      ID=c.CustomerID,
                      Name=c.CompanyName,
                      City=c.City,
                      State=c.Region
                   };
```

`Customers` — типизированная таблица LINQ (`System.Data.Linq.Table<Customer>`), которая подобна типизированной коллекции объектов `Customer` (вроде `List<Table>`), но предназначена для LINQ to SQL и заполняется из базы данных автоматически. Она реализует интерфейсы `IEnumerable/IQueryable`, что дает возможность ее использовать в качестве источника данных для LINQ в конструкции `from`, как любую коллекцию или массив.

Конструкция `where` ограничивает результаты только заказчиками из США. Конструкция `select` — проекция, подобная примерам из предыдущей главы, которая создает новый объект с членами `ID`, `Name`, `City` и `State`. Поскольку известно, что в результате будут только заказчики из США, можно переименовать `Region` в `State` для более точного отображения. Наконец, создается стандартный цикл `foreach`, подобный тому, что применялся в главе 23:

```
foreach (var item in queryResults) {
   Console.WriteLine(item);
};
```

В этом коде используется сгенерированный метод по умолчанию `ToString()` для каждого `item`, который позволяет сформатировать вывод `Console.WriteLine(item)`, чтобы можно было видеть значения для каждого проектированного члена экземпляра в фигурных скобках:

```
{ ID=WHITC, Name=White Clover Markets, City=Seattle, State=WA }
```

Наконец, пример завершается кодом приостановки вывода, чтобы можно было просмотреть результат:

```
Console.WriteLine("Press Enter/Return to continue..");
Console.ReadLine();
```

Таким образом, создан базовый запрос LINQ to SQL, который можно использовать в качестве основы для построения более сложных запросов.

Навигация по отношениям в базе данных

Одним из наиболее мощных аспектов ADO.NET Entity Framework является его способность автоматически создавать объекты LINQ to SQL, чтобы помочь выполнять навигацию по отношениям между связанными таблицами базы данных. В следующем практическом занятии вы добавите в класс LINQ to Entities связанную таблицу, напишете код навигации по взаимосвязанным объектам базы данных и выведете на консоль их значения.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Навигация по отношениям LINQ to Entities

Для создания примера в Visual C# 2010 выполните перечисленные ниже шаги.

1. Модифицируйте проект предыдущего примера `BegVCSharp_24_1_FirstLINQtoDataQuery` в каталоге `C:\BegVCSharp\Chapter24`, как описано ниже.
2. Откройте файл `Program.cs`. В методе `Main()` добавьте поле `Orders` к конструкции `select` запроса LINQ (не забудьте добавить запятую после `c.Region`, чтобы отделить добавленное поле от остальной части списка):

```
static void Main(string[] args)
{
   NorthwindDataContext northWindDataContext = new NorthwindDataContext();
   var queryResults = from c in northWindDataContext.Customers
                      where c.Country == "USA"
                      select new {
```

```
                                ID=c.CustomerID,
                                  Name=c.CompanyName,
                                  City=c.City,
                                  State=c.Region,
                                  Orders=c.Orders
                            };
```

3. Модифицируйте конструкцию `foreach` для вывода результатов запроса, как показано ниже:

```
foreach (var item in queryResults) {
    Console.WriteLine(
       "Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date",
          item.Name, item.City, item.State, item.Orders.Count
       );
       foreach (Order o in item.Orders) {
          Console.WriteLine("\t\t{0}\t{1}", o.OrderID, o.OrderDate);
       }
    };
    Console.WriteLine("Press Enter/Return to continue..");
    Console.ReadLine();
}
```

Фрагмент кода ***BegVCSharp\Chapter24\BegVCSharp_24_2_NavigatingDatabaseRelationships\Program.cs***

4. Скомпилируйте и запустите программу (можно просто нажать <F5> для запуска отладки). Вы увидите информацию о заказчиках из США вместе с их заказами (здесь приведена заключительная часть вывода):

```
Customer: Trail's Head Gourmet Provisioners Kirkland, WA
3 orders:    Order ID        Order Date
             10574    6/19/1997 12:00:00 AM
             10577    6/23/1997 12:00:00 AM
             10822    1/8/1998 12:00:00 AM
Customer: White Clover Markets Seattle, WA
14 orders:   Order ID        Order Date
             10269    7/31/1996 12:00:00 AM
             10344    11/1/1996 12:00:00 AM
             10469    3/10/1997 12:00:00 AM
             10483    3/24/1997 12:00:00 AM
             10504    4/11/1997 12:00:00 AM
             10596    7/11/1997 12:00:00 AM
             10693    10/6/1997 12:00:00 AM
             10696    10/8/1997 12:00:00 AM
             10723    10/30/1997 12:00:00 AM
             10740    11/13/1997 12:00:00 AM
             10861    1/30/1998 12:00:00 AM
             10904    2/24/1998 12:00:00 AM
             11032    4/17/1998 12:00:00 AM
             11066    5/1/1998 12:00:00 AM
Press Enter/Return to continue...
```

Как и ранее, нажмите клавишу <Enter> для завершения программы и закрытия консольного окна.

Описание работы

Поскольку в примере была модифицирована предыдущая программа, а не создана новая с нуля, не пришлось повторять все шаги для создания файла исходных данных `Model1.edmx` (обратите внимание, что примеры кода находятся в разных проектах, причем каждый с собственным экземпляром `Model1.edmx`).

В результате отметки таблицы Orders в диалоговом окне Choose Your Database Objects к исходному файлу Model1.edmx добавляется класс Order для представления таблицы Orders в отображении базы данных Northwind.

Среда Visual Studio 2010 обнаружила в базе данных отношение между таблицами Customers и Orders, поэтому добавила в класс Customer новый член — коллекцию Orders для представления отношения. Все это было сделано автоматически при добавлении новых элементов управления на форму.

В конструкцию запроса select был добавлен новый доступный член Orders:

```
select new {
            ID=c.CustomerID,
            Name=c.CompanyName,
            City=c.City,
            State=c.Region,
            Orders=c.Orders
           };
```

Orders — это специальный типизированный набор LINQ (System.Data.Linq.EntitySet<Order>), представляющий отношение между двумя таблицами в реляционной базе данных. Он реализует интерфейсы IEnumerable/IQueryable, чтобы его можно было использовать в качестве источника данных LINQ либо выполнять итерацию с помощью оператора foreach, как с любой коллекцией или массивом.

Подобно объекту Table, показанному в предыдущем примере, EntitySet похож на типизированную коллекцию объектов Order (вроде List<Order>), но в члене EntitySet конкретного экземпляра Customer появятся только те заказы, что размещены конкретным заказчиком.

Объекты Order в члене EntitySet объекта Customer соответствуют строкам заказов в базе данных, имеющим тот же самый идентификатор CustomerID, что и у заказчика.

Навигация по отношению просто предусматривает построение вложенного оператора foreach для итерации по всем заказчикам и затем — по их заказам:

```
foreach (var item in queryResults) {
   Console.WriteLine(
      "Customer: {0} {1}, {2}\n{3} orders:\tOrder ID\tOrder Date",
         item.Name, item.City, item.State, item.Orders.Count
   );
   foreach (Order o in item.Orders) {
      Console.WriteLine("\t\t{0}\t{1}", o.OrderID, o.OrderDate);
   }
};
```

Вместо простого использования форматирования ToString() по умолчанию вы форматируете вывод для лучшей читабельности, поэтому можно корректно отразить иерархию, с выводом списка заказов под каждым заказчиком. Форматная строка "Customer: {0} {1}, {2}\n{3} orders:\tOrder Date" включает места подстановки имени, города и штата каждого заказчика в первой строке, затем в следующей строке печатает заголовок столбца заказов данного заказчика. С использованием агрегатного метода LINQ по имени Count() выводится количество заказов каждого заказчика, после чего во вложенном операторе foreach выводятся идентификатор и дата заказа в каждой новой строке:

```
Customer: White Clover Markets Seattle, WA
14 orders:  Order ID           Order Date
            10269          7/31/1996 12:00:00 AM
            10344          11/1/1996 12:00:00 AM
```

Это форматирование несколько неаккуратно, поскольку вы видите время заказа, хотя на самом деле нужна только дата.

Итак, мы успешно запросили информацию из базы данных, и теперь наступил момент поработать с источником данных иного рода — XML.

Использование LINQ вместе с XML

Технология LINQ to XML не предназначена для замены стандартных API-интерфейсов XML, таких как XML DOM (Document Object Model — объектная модель документов), XPath, XQuery, XSLT и т.д. Если вы знакомы с этими API-интерфейсами или нуждаетесь либо изучаете их, стоит продолжать это делать.

LINQ to XML дополняет эти стандартные классы XML и облегчает работу с XML. Эта технология предоставляет дополнительные опции для создания и опроса данных XML, в результате чего упрощается код и ускоряется разработка во многих распространенных ситуациях, особенно в случае применения LINQ в других программах.

Функциональные конструкторы LINQ to XML

Как было показано в предыдущих главах, одной из характерных особенностей C# является облегчение конструирования объектов, с помощью таких средств, как инициализаторы объектов и анонимные типы. В LINQ to XML эта тенденция продолжается и предлагается способ создания документов XML, именуемый *функциональным конструированием*, при котором вызовы конструктором могут быть вложены так, чтобы естественным образом отражать структуру документа XML. В следующем практическом занятии функциональные конструкторы применяются для создания простого документа XML, содержащего заказчиков и заказы.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Конструкторы LINQ to XML

Для создания примера в Visual Studio 2010 выполните следующие шаги.

1. Создайте новое консольное приложение по имени `BegVCSharp_24_3_LinqToXmlConstructors` в каталоге `C:\BegVCSharp\Chapter24`.
2. Откройте файл `Program.cs`.
3. Добавьте в начало `Program.cs` ссылку на пространство имен `System.Xml.Linq`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

4. Добавьте следующий код в метод `Main()` внутри `Program.cs`:

```
static void Main(string[] args)
{
  XDocument xdoc = new XDocument(
     new XElement("customers",
     new XElement("customer",
     new XAttribute("ID", "A"),
     new XAttribute("City", "New York"),
     new XAttribute("Region", "North America"),
     new XElement("order",
     new XAttribute("Item", "Widget"),
     new XAttribute("Price", 100)
   ),
   new XElement("order",
     new XAttribute("Item", "Tire"),
     new XAttribute("Price", 200)
   )
  ),
```

```
        new XElement("customer",
            new XAttribute("ID", "B"),
            new XAttribute("City", "Mumbai"),
            new XAttribute("Region", "Asia"),
            new XElement("order",
                new XAttribute("Item", "Oven"),
                new XAttribute("Price", 501)
              )
            )
          )
        );
        Console.WriteLine(xdoc);
        Console.Write("Program finished, press Enter/Return to continue:");
        Console.ReadLine();
    }
```

Фрагмент кода ***BegVCSharp\Chapter24\BegVCSharp_24_3_LinqToXmlConstructors\Program.cs***

5. Скомпилируйте и выполните программу (можно просто нажать <F5> для запуска отладки). После этого вы увидите следующий вывод:

```
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

Этот документ XML содержит очень упрощенную версию данных о заказчиках и заказах, которые были показаны в предыдущем примере. Обратите внимание, что корневым элементом документа XML является `<customers>`, содержащий два вложенных элемента `<customer>`. Они, в свою очередь, содержат ряд вложенных элементов `<order>`. Элементы `<customer>` имеют два атрибута — `<City>` и `<Region>`, а элементы `<order>` — атрибуты `<Item>` и `<Price>`.

Нажмите клавишу <Enter> для выхода из программы и закрытия окна консоли. Если использовалась комбинация клавиш <Ctrl+F5> (запуск без отладки), может понадобиться нажать <Enter> два раза.

Описание работы

Первый шаг — установка ссылки на пространство имен `System.Xml.LInq`. Все прочие примеры в этой главе требуют добавления следующей строки в программу:

```
using System.Xml.Linq;
```

В то время как пространство имен `System.Linq` включается по умолчанию при создании проекта, пространство имен `System.Xml.Linq` должно быть включено за счет явного добавления строки кода.

Затем идут вызовы конструкторов LINQ to XML с именами `XDocument()`, `XElement()` и `Xattribute()`, которые вложены друг в друга, как показано ниже:

```
XDocument xdoc = new XDocument(
  new XElement("customers",
    new XElement("customer",
      new XAttribute("ID", "A"),
      ...
```

Обратите внимание, что сам этот код выглядит подобно XML: в нем документ содержит элементы, и каждый элемент содержит атрибуты и прочие элементы. Давайте рассмотрим каждый из этих конструкторов по очереди.

- `XDocument()`. Объект самого верхнего уровня в иерархии конструкторов LINQ to XML, который представляет документ XML целиком. Он выглядит в коде примера следующим образом:

```
static void Main(string[] args)
{
    XDocument xdoc = new XDocument(
...
);
```

  Список параметров `XDocument()` в этом фрагменте кода опущен, так что можно увидеть, где начинается и заканчивается вызов `XDocument()`. Подобно конструкторам LINQ to XML, `XDocument()` принимает массив объектов (`object[]`) в качестве одного из параметров, так что ему можно передавать множество других объектов, созданных другими конструкторами. Все прочие конструкторы, которые вызываются в этой программе, являются параметрами одного вызова конструктора `XDocument()`. Первый (и единственный) параметр, передаваемый в этой программе — конструктор `XElement()`.

- `XElement()`. Документ XML должен иметь корневой элемент, так что в большинстве случаев список параметров `XDocument()` начинается с объекта `Xelement`. Конструктор `XDocument()` принимает имя элемента в виде строки, за которой следует список объектов XML, содержащихся внутри этого элемента. Здесь корневым элементом является `"customers"`, который, в свою очередь, содержит список элементов `"customer"`:

```
new XElement("customers",
    new Xelement("customer",
    ...
    ),
...
)
```

  Элемент `"customer"` не содержит никаких других элементов XML. Вместо этого в нем определены три атрибута XML, которые создаются с помощью конструкторов `XAttribute()`.

- `XAttribute()`. Ниже к элементу `"customer"` добавляются три атрибута XML с именами `"ID"`, `"City"` и `"Region"`:

```
new XAttribute("ID", "A"),
new XAttribute("City", "New York"),
new XAttribute("Region", "North America"),
```

  Поскольку атрибут XML является по определению листовым узлом XML, не содержащим других узлов XML, конструктор `XAttribute()` принимает в качестве параметров только имя атрибута и его значение. В этом случае сгенерированы следующие три атрибута: `ID="A"`, `City="New York"` и `Region="North America"`.

- Другие конструкторы LINQ to XML. Существуют и другие конструкторы LINQ to XML (в примере программы они не вызываются) для всех типов узлов XML, такие как `XDeclaration()` для объявления XML в начале документа XML, `XComment()` — для всех комментариев XML и т.д. Эти прочие конструкторы делают возможным тонкий контроль над форматированием документа XML.

В завершение объяснения первого примера отметим, что вы добавили два дочерних элемента "order" к элементу "customer", за которым следуют атрибуты "ID", "City" и "Region":

```
new XElement("order",
   new XAttribute("Item", "Widget"),
   new XAttribute("Price", 100)
),
new XElement("order",
   new XAttribute("Item", "Tire"),
   new XAttribute("Price", 200)
)
```

Элементы заказов имеют атрибуты "ITem" и "Price", но не содержат других дочерних элементов.

Затем содержимое XDocument выводится на консоль:

```
Console.WriteLine(xdoc);
```

На экране консоли отображается текст документа XML с использованием метода ToString() по умолчанию элемента XDocument.

Наконец, вывод на консоль приостанавливается и ожидается нажатия пользователем клавиши <Enter>:

```
Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
```

После этого происходит выход из метода Main(), что завершает программу.

Конструирование текста элемента XML со строками

В только что рассмотренном примере XML-код форматировался без текстового содержимого элементов. Часто XML-код нуждается в текстовом содержимом и это очень легко сделать с помощью конструктора LINQ to XML по имени XElement(). Например, чтобы сделать идентификатор элемента <customer> текстом вместо атрибута, просто передайте строку в качестве параметра конструктору XElement, а не вложенного XAttribute:

```
XDocument xdoc = new XDocument(
   new XElement("customers",
      new XElement("customer",
      "AAAAAA",
      new XAttribute("City", "New York"),
      new XAttribute("Region", "North America")
   ),
   new XElement("customer",
      "BBBBBB",
      new XAttribute("City", "Mumbai"),
      new XAttribute("Region", "Asia")
   )
);
```

Этот код производит документ XML, который выглядит следующим образом:

```
<customers>
  <customer City="New York" Region="North America">AAAAAA</customer>
  <customer City="Mumbai" Region="Asia">BBBBBB</customer>
</customers>
```

Конструктор XElement() выполняет конкатенацию всех строк в списке параметров в текстовый раздел элемента.

Сохранение и загрузка документа XML

Возможно, вы заметили, что когда документ XML отображается на экране консоли с помощью `Console.WriteLine()`, то не показывается нормальное объявление XML, начинающееся с `<?xml version="1.0"`. Хотя можно создать такое объявление явно конструктором `XDeclaration()`, обычно делать это не нужно, поскольку оно создается автоматически при сохранении документа XML в файл методом LINQ to XML по имени `Save()`.

Вдобавок, хотя конструирование документов XML в программе полезно для понимания работы конструкторов, делать это придется нечасто. Гораздо чаще вы будете загружать документы XML из внешнего источника, такого как файл.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Сохранение и загрузка документа XML

Выполните следующие шаги для создания примера в Visual Studio 2010.

1. Модифицируйте предыдущий пример либо создайте новое консольное приложение по имени `BegVCSharp_24-4-SaveLoadXML` в каталоге `C:\BegVCSharp\Chapter24`.
2. Откройте файл `Program.cs`.
3. Добавьте ссылку на пространство имен `System.Xml.Linq` в начало `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

   Если вы модифицируете предыдущий пример, ссылка уже должна быть на месте.

4. Если его еще нет, добавьте в метод `Main()` внутри файла `Program.cs` конструктор документа XML, его вложенный элемент XML и вызовы атрибутов из предыдущего примера:

```
static void Main(string[] args)
{
  XDocument xdoc = new XDocument(
    new XElement("customers",
      new XElement("customer",
      new XAttribute("ID", "A"),
      new XAttribute("City", "New York"),
      new XAttribute("Region", "North America"),
      new XElement("order",
        new XAttribute("Item", "Widget"),
        new XAttribute("Price", 100)
      ),
      new XElement("order",
        new XAttribute("Item", "Tire"),
        new XAttribute("Price", 200)
      )
    ),
    new XElement("customer",
      new XAttribute("ID", "B"),
      new XAttribute("City", "Mumbai"),
      new XAttribute("Region", "Asia"),
      new XElement("order",
        new XAttribute("Item", "Oven"),
        new XAttribute("Price", 501)
      )
    )
  )
);
```

Фрагмент кода ***BegVCSharp\Chapter24\BegVCSharp_24_4_SaveLoadXml\Program.cs***

5. После добавления конструктора документа на предыдущем шаге добавьте в конец метода `Main()` внутри файла `Program.cs` следующий код для сохранения, загрузки и отображения документа XML:

```
    string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example2.xml";
    xdoc.Save(xmlFileName);
    XDocument xdoc2 = Xdocument.Load(xmlFileName);
    Console.WriteLine("Contents of xdoc2:");
    Console.WriteLine(xdoc2);
    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

6. Скомпилируйте и запустите программу (можно просто нажать <F5> для запуска отладки). После этого в консольном окне должен появиться следующий вывод:

```
Contents of xdoc2:
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

Нажмите клавишу <Enter> для завершения программы и закрытия консольного экрана. Если для запуска использовалась комбинация клавиш <Ctrl+F5> (запуск без отладки), возможно, <Enter> понадобится нажать два раза.

Описание работы

Как и ранее, первый шаг состоит в установке ссылки на пространство имен `System.Xml.Linq`. Затем следуют вложенные вызовы конструкторов LINQ to XML по именам `XDocument()`, `XElement()` и `Xattribute()`. Объяснение этих частей и прочего кода, повторяемого из первого примера, ищите в описании работы первого примера.

После создания объекта `XDocument` указывается имя файла в виде строки и документ XML сохраняется в файле вызовом метода `Save()`:

```
string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example2.xml";
xdoc.Save(xmlFileName);
```

Хотя в конкретном случае документ сохраняется в файле с указанным именем, метод `Save()` также перегружен для сохранения в `System.IO.TextWriter` или `System.Xml.XmlWriter`, что может подойти, если вы пишете другую программу, в которой уже используется один из этих классов для записи в файл.

Метод `Save()` также имеет перегрузку, в которой можно указывать опции сохранения (`SaveOptions`) для отключения форматирования (по умолчанию документ XML сохраняется с отступами и пробелами, что придает им симпатичный внешний вид).

После сохранения документа в файле он загружается в новый экземпляр `XDocument` по имени `xdoc2`:

```
XDocument xdoc2 = XDocument.Load(xmlFileName);
```

Метод `XDocument.Load()` является статическим, потому что это метод типа фабрики, создающей новый экземпляр `XDocument`; его можно применять для загрузки документа, созданного совершенно другой программой.

Затем документ отображается, как это делалось ранее, но на этот раз с использованием экземпляра xdoc2, загруженного из файла. Остальная часть программы совпадает с соответствующей частью из предыдущего примера:

```
Console.WriteLine("Contents of xdoc2:");
Console.WriteLine(xdoc2);
Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
```

Загрузка XML из строки

Иногда вместо загрузки документа XML из файла вы получаете XML-код от другого приложения в виде строки, через один или более методов. Для создания документов XML из строк в LINQ to XML применяется метод Parse():

```
XDocument xdoc = XDocument.Parse(@"
       <customers>
            <customer ID=""A"" City=""New York"" Region=""North America"">
              <order Item=""Widget"" Price=""100"" />
              <order Item=""Tire"" Price=""200"" />
            </customer>
            <customer ID=""B"" City=""Mumbai"" Region=""Asia"">
               <order Item=""Oven"" Price=""501"" />
            </customer>
       </customers>
       ");
```

Это дает тот же результат, что и загрузка документа из файла. Как и Load(), Parse() — метод уровня класса, создающий новый экземпляр XDocument; конструировать новый объект XDocument перед вызовом метода Parse() не нужно.

Хотя строковый литерал XML в предыдущем примере содержит удвоенные символы кавычек (""), в действительном содержимом строки они не удваиваются. Удвоенные символы — всего лишь соглашение для включения кавычек в не защищенный с помощью @ строковый литерал.

Содержимое сохраненного документа XML

Воспользуйтесь браузером Internet Explorer для открытия документа, сохраненного в предыдущем примере. Укажите полное путевое имя C:\BegVCSharp\Chapter24\Xml\example2.xml в поле адреса.

Обратите внимание, что объявление документа XML <?xml version="1.0" encoding="utf-8" ?> появляется в начале сохраненного документа, даже несмотря на то, что оно не отображается, когда содержимое объекта XDocument просто выводится на экран с помощью Console.WriteLine(). Применяя умолчания, предоставляемые LINQ to XML, не нужно беспокоиться об объявлении и многих других деталях XML.

Кодировкой по умолчанию для документов XML в Windows является UTF-8 (8-битный формат Unicode Transformation Format). Это не должно изменяться кроме самых необычных ситуаций, таких как создание ASCII-кодированного XML-документа, который должен использоваться другой программой, не воспринимающей UTF-8. В этом случае можно либо добавить в начало списка параметров конструктора XDocument() объект XDeclaration() с кодировкой, установленной в ASCII, либо соответствующим образом установить свойство Declaration объекта Xdocument:

```
xdoc.Declaration = new Xdeclaration("1.0", "us-ascii", "yes");
```

Работа с фрагментами XML

В отличие от некоторых API-интерфейсов XML, LINQ to XML работает с фрагментами XML (частичными или неполными документами XML) почти так же, как с полными документами XML. Имея дело с фрагментами, вы вместо `XDocument` просто работаете с `XElement` в качестве объекта верхнего уровня.

Единственное ограничение, которое здесь действует, касается невозможности добавлять некоторые из самых экзотических типов узлов XML, применимых только к документам XML или фрагментам XML. Примерами таких узлов могут быть `XComment` для комментариев XML, `XDeclaration` для объявления документа XML и `XProcessingInstruction` для инструкций обработки XML.

В следующем практическом занятии мы загрузим, сохраним и будем манипулировать элементом XML и его дочерними узлами, как ранее делалось с документом XML.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с фрагментами XML

Выполните следующие шаги для создания примера в Visual Studio 2010.

1. Модифицируйте предыдущий пример или создайте новое консольное приложение по имени `BegVCSharp_24-5-XMLFragments` в каталоге `C:\BegVCSharp\Chapter24`.
2. Откройте файл `Program.cs`.
3. Добавьте ссылку на пространство имен `System.Xml.Linq` в начало `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

   Если вы модифицируете предыдущий пример, это уже должно быть на месте.

4. Добавьте в метод `Main()` внутри `Program.cs` элемент XML, не содержащий конструктора документа XML, который использовался в предыдущих примерах:

```
static void Main(string[] args)
{
    XElement xcust =
        new XElement("customers",
            new XElement("customer",
            new XAttribute("ID", "A"),
            new XAttribute("City", "New York"),
            new XAttribute("Region", "North America"),
            new XElement("order",
                new XAttribute("Item", "Widget"),
                new XAttribute("Price", 100)
            ),
        new XElement("order",
            new XAttribute("Item", "Tire"),
            new XAttribute("Price", 200)
        )
    ),
    new XElement("customer",
        new XAttribute("ID", "B"),
        new XAttribute("City", "Mumbai"),
        new XAttribute("Region", "Asia"),
```

```
        new XElement("order",
            new XAttribute("Item", "Oven"),
            new XAttribute("Price", 501)
        )
    )
);
```

*Фрагмент кода **BegVCSharp\Chapter24\BegVCSharp_24_5_XmlFragments\Program.cs***

5. После конструктора элемента XML, добавленного на предыдущем шаге, поместите следующий код для сохранения, загрузки и отображения элемента XML:

```
    string xmlFileName = @"c:\BegVCSharp\Chapter24\Xml\example3.xml";
    xcust.Save(xmlFileName);
    XElement xcust2 = XElement.Load(xmlFileName);
    Console.WriteLine("Contents of xcust:");
    Console.WriteLine(xcust);
    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

6. Скомпилируйте и выполните программу (можно просто нажать <F5> для запуска отладки). В окне консоли должен появиться следующий вывод:

```
Contents of XElement xcust2:
<customers>
  <customer ID="A" City="New York" Region="North America">
    <order Item="Widget" Price="100" />
    <order Item="Tire" Price="200" />
  </customer>
  <customer ID="B" City="Mumbai" Region="Asia">
    <order Item="Oven" Price="501" />
  </customer>
</customers>
Program finished, press Enter/Return to continue:
```

Нажмите клавишу <Enter> для завершения программы и закрытия экрана консоли. Если для запуска использовалась комбинация клавиш <Ctrl+F5> (запуск без отладки), возможно, <Enter> понадобится нажать два раза.

Описание работы

Как `XElement`, так и `XDocument` унаследованы от класса LINQ to XML по имени `XContainer`, который реализует узел XML, способный содержать в себе другие узлы. Оба класса также реализуют методы `Load()` и `Save()`, так что большинство операций, которые могут выполняться в `XDocument()` в LINQ to XML, также могут выполняться на экземпляре `XElement` и его наследниках.

Вы просто создаете экземпляр `XElement`, имеющий ту же структуру, что и `XDocument`, применяемый в предыдущих примерах, но без включения `XDocument`. Все операции для этой конкретной программы работают также с фрагментом `XElement`.

`XElement` также поддерживает методы `Load()` и `Save()` для загрузки XML-кода соответственно из файлов и строк.

Генерация документов XML из баз данных

XML часто используется для передачи данных между клиентской и серверной машинами, либо между уровнями в многоуровневом приложении. Довольно часто приходится запрашивать какую-то информацию из базы данных и затем строить документ XML или фрагмент данных для передачи на другой уровень. В следующем практическом занятии мы создадим запрос для нахождения информации в базе данных примеров Northwind, приме-

ним LINQ to SQL для запроса данных, а затем с помощью классов LINQ to XML преобразуем данные в XML.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Генерация документов XML из базы данных

Выполните следующие шаги для создания примера в Visual Studio 2010.

1. Создайте новое консольное приложение по имени `BegVCSharp_24_6_XMLfromDatabase` в каталоге `C:\BegVCSharp\Chapter24`.
2. Как описано в примере "Первый запрос LINQ к базе данных" в начале этой главы, добавьте к проекту новый источник данных по имени `Model1.edmx` и соединение с базой данных примеров Northwind.
3. Скомпилируйте программу, чтобы классы и свойства, определенные в `Model1.edmx`, были доступны через IntelliSense для редактирования кода на последующих шагах.
4. Откройте файл `Program.cs`.
5. Добавьте ссылку на пространство имен `System.Xml.Linq` в начало `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

6. Добавьте следующий код в метод `Main()` внутри `Program.cs`:

```
static void Main(string[] args)
{
    NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
    XElement northwindCustomerOrders =
    new XElement("customers",
      from c in northWindEntities.Customers.AsEnumerable()
      select new XElement("customer",
        new XAttribute("ID", c.CustomerID),
        new XAttribute("City", c.City),
        new XAttribute("Company", c.CompanyName),
           from o in c.Orders
           select new XElement("order",
             new XAttribute("orderID", o.OrderID),
             new XAttribute("orderDay",
               o.OrderDate.Value.Day),
             new XAttribute("orderMonth",
               o.OrderDate.Value.Month),
             new XAttribute("orderYear",
               o.OrderDate.Value.Year),
             new XAttribute("orderTotal",
               o.Order_Details.Sum(od => od.Quantity * od.UnitPrice))
           ) // конец order
      ) // конец customer
    ); // конец customers
    string xmlFileName =
           @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
    northwindCustomerOrders.Save(xmlFileName);
    Console.WriteLine("Successfully saved Northwind customer orders to:");
    // Список заказов для заказчика из Northwind успешно сохранен
    Console.WriteLine(xmlFileName);
    Console.Write("Program finished, press Enter/Return to continue:");
    Console.ReadLine();
}
```

Фрагмент кода ***BegVCSharp\Chapter24\BegVCSharp_24_6_XMLfromDatabase\Program.cs***

7. После компиляции и запуска программы (можно просто нажать <F5> для запуска отладки), в окне консоли должен появиться следующий вывод:

```
Successfully saved Northwind customer orders to:
C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml
Program finished, press Enter/Return to continue:
```

Просто нажмите клавишу <Enter> для завершения программы и закрытия экрана консоли. Если для запуска использовалась комбинация клавиш <Ctrl+F5> (запуск без отладки), возможно, <Enter> понадобится нажать два раза.

Описание работы

В `Program.cs` была добавлена ссылка на пространство имен `System.Xml.Linq`, чтобы можно было вызывать классы конструкторов LINQ to XML.

Как было описано в первой части этой главы, вы создали источник данных для базы примеров Northwind и затем воспользовались Visual Studio 2010 для создания объектной модели LINQ to Entities данных Northwind. В главной программе был создан экземпляр класса контекста данных Northwind для работы со следующим отображением:

```
NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
```

Запрос LINQ to Entities использует член `Customers` контекста данных Northwind в качестве источника данных и проходит вниз, через таблицы `Customers`, `Orders` и `Order Details`, для генерации списка всех заказов, размещенных заказчиком. Однако, по причине отложенного выполнения в LINQ to Entities, с помощью метода `AsEnumerable()` объекта `Customer` промежуточный результат был преобразован в расположенный в памяти перечислимый тип LINQ to Objects. Наконец, результат запроса в конструкции `select` был спроектирован во вложенный набор элементов и атрибутов LINQ to XML.

```
XElement northwindCustomerOrders =
    new XElement("customers",
     from c in northWindDataContext.Customers.AsEnumerable()
     select new XElement("customer",
       new XAttribute("ID", c.CustomerID),
       new XAttribute("City", c.City),
       new XAttribute("Company", c.CompanyName),
       from o in c.Orders
          select new XElement("order",
            new XAttribute("orderID", o.OrderID),
            new XAttribute("orderDay",
              o.OrderDate.Value.Day),
            new XAttribute("orderMonth",
              o.OrderDate.Value.Month),
            new XAttribute("orderYear",
              o.OrderDate.Value.Year),
            new XAttribute("orderTotal",
              o.Order_Details.Sum(od => od.Quantity * od.UnitPrice))
          ) // конец order
      ) // конец customer
); // конец customers
```

Чтобы собрать все заказы заказчика, используется второй запрос LINQ (`from o in c.Orders...`), вложенный внутрь первого (`from c in northwindDatacontext.Customers...`).

Для упрощения запроса XML поле `OrderDate` разделено на компоненты — месяц, день и год; в следующем примере будет показано, как это используется.

Наконец, сгенерированный документ XML сохраняется в файл, как было показано в предыдущем примере:

```
string xmlFileName =
      @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
northwindCustomerOrders.Save(xmlFileName);
Console.WriteLine("Successfully saved Northwind customer orders to:");
Console.WriteLine(xmlFileName);
```

Теперь давайте напишем запрос к файлу XML, сохраненному на диске.

Отправка запросов к документу XML

Зачем вообще может понадобиться выполнять запросы LINQ в отношении документов XML? Если программа принимает документ XML, сгенерированный другой программой, может понадобиться отыскать специфические элементы или атрибуты XML внутри полученного документа, чтобы определить, как его обрабатывать. Программе может требоваться только некоторое подмножество содержимого XML. Возможно, необходимо подсчитать количество элементов внутри документа либо искать элементы или атрибуты, удовлетворяющие определенным условиям. Запросы LINQ предлагают мощное решение для подобного рода ситуаций.

Для опроса документа XML такие классы LINQ to XML, как `XDocument` и `XElement`, поддерживают свойства и методы, которые возвращают коллекции объектов LINQ to XML, содержащиеся внутри документа XML или в фрагменте, представленном этим классом LINQ to XML.

В следующем практическом занятии применяются опрашиваемые методы и свойства в отношении документа XML, созданного в предыдущем примере.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Опрос документа XML

Для создания примера в Visual Studio 2010 выполните следующие шаги.

1. Создайте новое консольное приложение по имени `BegVCSharp_24-7-QueryXML` в каталоге `C:\BegVCSharp\Chapter24`.
2. Откройте файл `Program.cs`.
3. Добавьте ссылку на пространство имен `System.Xml.Linq` в начало `Program.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
```

4. Добавьте следующий код в метод `Main()` внутри `Program.cs`:

```
static void Main(string[] args)
{
  string xmlFileName = @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
  XDocument customers = XDocument.Load(xmlFileName);
  Console.WriteLine("Elements in loaded document:");
  // Список элементов в загруженном документе
  var queryResult = from c in customers.Elements() select c.Name;
  foreach (var item in queryResult)
  {
    Console.WriteLine(item);
  }
  Console.Write("Press Enter/Return to continue:");
  Console.ReadLine();
}
```

*Фрагмент кода **BegVCSharp\Chapter24\BegVCSharp_24_7_QueryXML\Program.cs***

5. Скомпилируйте и запустите программу (можно просто нажать <F5> для запуска отладки). Вы увидите следующий вывод:

```
Elements in loaded document:
customers
Press Enter/Return to continue:
```

6. Нажмите клавишу <Enter> для завершения программы и закрытия экрана консоли. Если для запуска использовалась комбинация клавиш <Ctrl+F5> (запуск без отладки), возможно, <Enter> понадобится нажать два раза.

Описание работы

По мере ознакомления с каждым методом запроса, только что созданный пример запроса LINQ to XML будет изменяться для использования нового метода. Каждый из этих запросов возвращает коллекцию элементов LINQ to XML или объектов атрибутов, которые имеют свойство `Name`, так что конструкция `select` просто возвращает значение этого свойства для вывода в цикле `foreach`:

```
var queryResult = from c in customers.Elements()
                  select c.Name;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

Подобного рода код обычно используется в начале разработки или при отладке, поскольку он позволяет увидеть, что возвращает запрос. После этого вывод изменяется для отображения бизнес-результатов, которые нужны конечным пользователям.

Использование членов запросов LINQ to XML

В этом разделе рассматриваются доступные члены запроса LINQ to XML. Все они будут по очереди опробованы с использованием файла `NorthwindCustomerOrders.xml` в качестве источника данных.

Метод `Elements()`

Первый метод запроса LINQ to XML, который будет рассматриваться — это `Elements()`, определенный в классе `XDocument`. Этот член также доступен в классе `XElement`.

Метод `Elements()` возвращает набор элементов первого уровня в документе XML или его фрагменте. В действительном документе XML, таком как в только что созданном нами файле, `NorthwindCustomerOrders.xml`, есть только один элемент первого уровня под названием `customers`:

```
<?xml version="1.0" encoding="utf-8" ?>
<customers>
...
</customers>
```

Все прочие элементы являются дочерними по отношению к `customers`, поэтому `Elements()` возвращает только один элемент:

```
Elements in loaded document:
customers
```

Фрагмент XML может содержать множество элементов первого уровня, но обычно требуется опрашивать дочерние элементы, что позволяет делать описанный ниже метод `Descendants()`.

Метод Descendants()

Следующий метод запроса LINQ to XML — Descendants(), являющийся членом класса XDocument. Он также доступен в классе XElement.

Метод Descendants() возвращает "плоский" список всех дочерних элементов (на всех уровнях) документа XML или его фрагмента. Модифицируйте пример BegVCSharp_24-7-QueryXML следующим образом:

```
Console.WriteLine("All descendants in document:");
// Список дочерних элементов в документе
queryResult =
   from c in customers.Descendants()
   select c.Name;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

Скомпилируйте и запустите код. Вы увидите имена элементов customer и orders, перечисленные в порядке, в котором они содержатся в документе:

```
All descendants in document:
customer
order
order
...
customer
order
...
customer
order
...
order
order
Press Enter/Return to continue:
```

Вывод прокручивается на экране, так что вы можете не увидеть его начало. Это отражает тот факт, что файл NorthwindCustomerOrders.xml ниже корневого элемента customers содержит только элементы customer и order:

```
<?xml version="1.0" encoding="utf-8" ?>
<customers>
<customer ... [{[SPACE]}]>
<order ... />
<order ... />
...
</customer>
<customer ...>
<order ... />
...
```

Чтобы сделать вывод более управляемым, можно добавить LINQ-операцию Distinct() к результирующей обработке:

```
Console.WriteLine("All distinct descendants in document:");
// Список уникальных дочерних элементов в документе
 var queryResult =
    from c in customers.Descendants()
    select c.Name;
foreach (var item in queryResult.Distinct())
```

В результате получается список только отличающихся имен элементов:

```
All distinct descendants in document:
customers
customer
order
Press Enter/Return to continue:
```

Это очень удобно для просмотра структуры документа, когда вы только начинаете работать с ним, однако нахождение всех элементов — не та проблема, которую приходится решать в реальных приложениях.

Более распространенный сценарий предполагает поиск элементов-наследников по определенному имени. Метод `Descendants()` имеет перегрузку, которая принимает имя желаемого элемента в качестве строкового параметра:

```
Console.WriteLine("Descendants named 'customer':");
// Дочерние элементы по имени customer
 var queryResult =
     from c in customers.Descendants("customer")
     select c.Name;
foreach (var item in queryResult) // вызов Distinct() удален
{
   Console.WriteLine(item);
}
```

Код вернет только элементы `customer`:

```
Descendants named 'customer':
customer
customer
customer
...
customer
customer
Press Enter/Return to continue:
```

Ясно, что такое запрос намного полезнее. Запрашивая список элементов известного типа, можно затем искать определенные атрибуты, что и будет рассмотрено далее.

Для полноты картины вам следует знать, что LINQ to XML также предоставляет метод `Ancestors()`, служащий дополнением метода `Descendants()`. Этот метод возвращает "плоский" список всех элементов, находящихся выше исходного элемента в древовидной структуре документа XML. Он используется не так часто, как `Descandants()`, потому что обработка документов XML обычно начинается от корня к листьям в дереве элементов и атрибутов. Более часто применяется свойство `Parent`, указывающее на единственного предка уровнем выше.

Метод `Attributes()`

Следующий метод запроса LINQ to XML из числа рассматриваемых — это `Attributes()`. Он возвращает все атрибуты текущего выбранного элемента. Чтобы увидеть, как работает этот метод, модифицируйте пример `BegVCSharp_24-7-QueryXML` следующим образом:

```
Console.WriteLine("Attributes of descendants named 'customer':");
// Атрибуты дочерних элементов по имени customer
var queryResult =
   from c in customers.Descendants("customer").Attributes()
   select c.Name;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

Скомпилируйте и запустите код. Вы должны увидеть имена атрибутов элементов customer:

```
Attributes of descendants named 'customer':
ID
City
Company
ID
City
Company
...
ID
City
Company
ID
City
Company
Press Enter/Return to continue:
```

И снова вывод прокручивается на экране. Этот запрос находит имена атрибутов элементов customer:

```
<customer ID= ... City= ... Company= ... >
<customer ID= ... City= ... Company= ... >
<customer ID= ... City= ... Company= ... >
...
```

Как и в Descendants(), методу Attributes() можно передать определенное имя для поиска. Вдобавок ограничиваться отображением имени не обязательно — можно также вывести и значение атрибута. Следующий запрос отображает атрибуты заказчика по имени Company:

```
Console.WriteLine("customer attributes named 'Company':");
// Атрибуты заказчика по имени Company
var queryResult =
   from c in customers.Descendants("customer").Attributes("Company")
   select c;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

Скомпилируйте и запустите код. Вы увидите значения атрибута Company для элементов customer:

```
...
Company="Toms SpezialitЁaten"
Company="Tortuga Restaurante"
Company="Tradiёcao Hipermercados"
Company="Trail's Head Gourmet Provisioners"
Company="Vaffeljernet"
Company="Victuailles en stock"
Company="Vins et alcools Chevalier"
Company="Die Wandernde Kuh"
Company="Wartian Herkku"
Company="Wellington Importadora"
Company="White Clover Markets"
Company="Wilman Kala"
Company="Wolski Zajazd"
Press Enter/Return to continue:
```

Ниже показан другой пример, на этот раз с элементами orders и атрибутом orderYear:

```
Console.WriteLine("order attributes named 'orderYear':");
// Атрибуты заказа по имени orderYear
var queryResult =
    from c in customers.Descendants("order").Attributes("orderYear")
    select c;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

Скомпилируйте и запустите код. Вывод будет выглядеть следующим образом:

```
...
orderYear="1998"
orderYear="1998"
orderYear="1998"
orderYear="1996"
orderYear="1997"
orderYear="1997"
orderYear="1998"
orderYear="1998"
orderYear="1998"
orderYear="1998"
Press Enter/Return to continue:
```

Можно также получить значение атрибута специально (в данном примере это год) через свойство Value:

```
Console.WriteLine("Values of order attributes named 'orderYear':");
// Значения атрибутов заказа по имени orderYear
var queryResult =
    from c in customers.Descendants("order").Attributes("orderYear")
    select c.Value;
foreach (var item in queryResult)
{
   Console.WriteLine(item);
}
```

Скомпилируйте и запустите код. На экране консоли должен появиться следующий вывод:

```
...
1996
1997
1997
1998
1998
1998
1998
Press Enter/Return to continue:
```

Теперь имеется возможность задавать специальные вопросы, например, в каком году начали поступать заказы? Для получения ответа можно воспользоваться тем же запросом, но применить агрегатную операцию Min() к результату вместо обычного цикла foreach:

```
var queryResult =
      from c in customers.Descendants("order").Attributes("orderYear")
      select c.Value;
Console.WriteLine("Earliest year in which orders were placed: {0}",
                                                queryResults.Min());
```

Скомпилируйте и запустите, чтобы увидеть ответ — 1996:

```
Earliest year in which orders were placed: 1996
Press Enter/Return to continue:
```

Более специфические запросы рассматриваются в упражнениях к этой главе.

Резюме

На этом рассмотрение вопросов использования LINQ с базами данных и XML завершено. Как вы видели, LINQ to XML интегрирует концепции LINQ с простым в применении альтернативным API-интерфейсом XML, который позволяет внедрять XML в другие программы, использующие LINQ. В результате запросы к документам XML становятся простыми и естественными для программистов, уже знакомых с LINQ в той или иной форме.

В этой главе вы узнали, как конструировать документы XML с помощью функциональных конструкторов LINQ to XML, а также как загружать и сохранять документы XML с применением LINQ to XML.

Вы научились использовать LINQ to XML при работе с неполными документами XML (фрагментами), а также генерировать XML-документы из запросов LINQ to SQL или LINQ to Objects.

Наконец, вы узнали, как опрашивать существующий документ XML посредством LINQ to XML, а также ознакомились с расширенными средствами LINQ, такими как агрегатные операции.

Упражнения

1. Создайте следующий документ XML с использованием конструкторов LINQ to XML:

```
<employees>
  <employee ID="1001" FirstName="Fred" LastName="Lancelot">
    <Skills>
      <Language>C#</Language>
      <Math>Calculus</Math>
    </Skills>
  </employee>
  <employee ID="2002" FirstName="Jerry" LastName="Garcia">
    <Skills>
      <Language>French</Language>
      <Math>Business</Math>
    </Skills>
  </employee>
</employees>
```

2. Напишите запрос к созданному файлу `NorthwindCustomerOrders.xml`, чтобы найти самых старых заказчиков (тех, что сделали заказы в первый год работы Northwind — 1996).
3. Напишите запрос к файлу `NorthwindCustomerOrders.xml`, чтобы найти заказчиков, которые сделали заказы на сумму более $10 000.
4. Напишите запрос к файлу `NorthwindCustomerOrders.xml`, чтобы найти время существования наиболее активных заказчиков, например, компаний, разместивших заказы на общую сумму более $100 000.
5. Воспользуйтесь LINQ to Entities для отображения детальной информации из таблиц `Products` и `Employees` в базе данных Northwind.
6. Создайте запрос LINQ to Entities для отображения наиболее продаваемых товаров из базы данных Northwind.
7. Создайте групповой запрос для отображения наиболее продаваемых товаров по странам.

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Ключевые концепции
Вариации LINQ	Каждый из различных источников данных в .NET имеет соответствующую вариацию LINQ, или “манеру”, в которой запрашиваются данные.
Отправка запросов к базе данных с помощью LINQ	Генерация класса LINQ to Entities для базы данных осуществляется с помощью мастера конфигурирования источников данных (Data Source Configuration Wizard) в Visual C# 2010 (доступного через пункт меню Data⇨Add New Data Source (Данные⇨Добавить новый источник данных)).
Навигация по отношениям в базе данных с помощью LINQ	Классы LINQ to Entities включают поддающиеся навигации члены для каждой связанной сущности данных (таблицы), которая добавляется к источнику данных.
Конструирование документов XML с помощью LINQ	LINQ to XML поддерживает очень мощные функциональные конструкторы для создания документов XML из запросов LINQ.
Создание документов XML из баз данных	Документы XML можно конструировать из баз данных, комбинируя в одном запросе LINQ to Entities, LINQ to Objects и LINQ to XML.
Создание файлов и фрагментов XML	LINQ to XML включает методы для загрузки и сохранения XML-кода в файлах, а также для простой манипуляции частями документов XML.

ЧАСТЬ V

Дополнительные технологии

В ЭТОЙ ЧАСТИ...

25

Windows Presentation Foundation

В ЭТОЙ ГЛАВЕ...

- Что собой представляет WPF
- Структура базового приложения WPF
- Основы WPF
- Программирование с использованием WPF

Ранее в этой книге демонстрировались приложения двух основных типов: настольные приложения, которые пользователи запускают непосредственно, и веб-приложения, доступные пользователю через браузер. Они создавались с помощью двух частей .NET Framework: Windows Forms и страницы ASP.NET. Упомянутые типы приложений обладают своими достоинствами и недостатками. В то время как настольные приложения более гибкие и отзывчивые, веб-приложения могут быть доступны удаленно сразу многим пользователям одновременно.

Однако в современной вычислительной среде границы между приложениями все более и более размываются.

С появлением веб-служб, а теперь — Windows Communication Foundation (см. главу 26), как настольные, так и веб-приложения могут работать в распределенном режиме, обмениваясь данными по локальным и глобальным сетям. Вдобавок клиентские веб-приложения (т.е. браузеры вроде Internet Explorer или Firefox) теперь не могут рассматриваться как "тонкие" клиенты, функциональность которых ограничена простым отображением информации. Современные браузеры и компьютеры, на которых они запускаются, способны на гораздо большее.

В последние годы произошел заметный сдвиг в сторону унификации пользовательского интерфейса. Теперь веб-приложения нередко используют такие технологии, как JavaScript, Flash, приложения Java и т.п., все больше напоминая настольные приложения. Чтобы удостовериться в этом, достаточно взглянуть, к примеру, на возможности Google Docs. И наоборот, настольные приложения становятся все более "подключенными", обладая средствами в диапазоне от простых (автоматические обновления, онлайновая справка и т.д.) до развитых (таких как онлайновые источники данных и одноранговые сетевые коммуникации). Это проиллюстрировано на рис. 25.1.

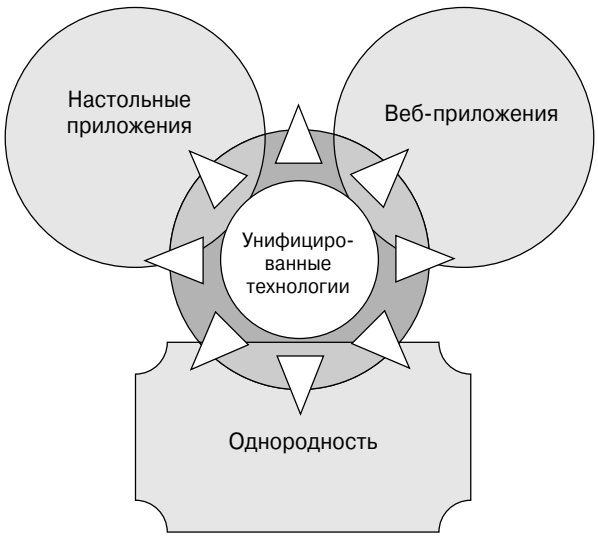


Рис. 25.1. *Сближение настольных и веб-приложений*

Windows Presentation Foundation (WPF) — унифицированная технология, позволяющая разрабатывать приложения, которые заполняют пробел между настольными и Интернет-приложениями. В этой главе будет показано, что приложения WPF могут выполняться как настольные либо как веб-приложения внутри браузера. Существует также усеченная версия WPF, называемая Silverlight, которую можно использовать для добавления динамического содержимого к веб-приложениям.

В этой главе вы узнаете о WPF и о том, как применять эту технологию для создания приложений следующего поколения.

Что собой представляет WPF

WPF — это технология, позволяющая разрабатывать независимые от платформы приложения с ясно определенной границей между дизайном и функциональностью. Она заимствует и развивает концепции и классы от многих предшествующих технологий, включая Windows Forms, ASP.NET, XML, технологии привязки данных, GDI+ и т.п. Если у вас есть опыт построения веб-приложений с помощью .NET Framework, то, взглянув на код приложения WPF, вы немедленно обнаружите эти сходства. В частности, в приложениях WPF используется разметка, а также модель отделенного кода, подобная той, что применяется в ASP.NET. Однако "за кулисами" остается множество различий и сходств, которые в сумме формируют совершенно новое впечатление у разработчиков и пользователей.

Одной из ключевых концепций разработки WPF является почти полное отделение дизайна от функциональности. Это отделение позволяет проектировщикам и разработчикам C# работать вместе в рамках одного проекта с определенной степенью свободы, которая ранее требовала развитых проектных решений и дополнительного инструментария. Эта функциональность была с благодарностью принята всеми — от небольших коллективов и разработчиков-любителей до огромных команд разработчиков и дизайнеров, задействованных в крупномасштабных проектах.

В следующих разделах будет показано, какие преимущества предлагает технология WPF дизайнерам и разработчикам при совместной их работе.

WPF для дизайнеров интерфейса

Язык, применяемый для построения пользовательского интерфейса в WPF, называется XAML (eXtensible Application Markup Language — расширяемый язык разметки приложений). Он похож на язык разметки, используемый в ASP.NET, в том отношении, что имеет синтаксис XML и позволяет добавлять к пользовательскому интерфейсу элементы в декларативной иерархической манере. То есть можно добавлять элементы управления в форме XML-элементов и указывать их свойства в атрибутах XML. Элементы управления могут также содержать другие элементы, что удобно как при разметке, так и в плане функциональности.

Однако XAML является намного более мощным языком, чем ASP.NET, и он не ограничивается возможностями HTML при визуализации отображения для пользователя. Язык XAML спроектирован в расчете на мощные современные графические карты и позволяет применять расширенные средства, которые поддерживают эти графические карты через DirectX 7 и последующие версии. Ниже перечислены некоторые из этих средств.

- Координаты с плавающей точкой и векторная графика для обеспечения компоновки, которая может масштабироваться, поворачиваться либо иным образом трансформироваться, без потери качества.
- Возможности расширенной двух- и трехмерной визуализации.
- Расширенная обработка и визуализация шрифтов.
- Сплошные, градиентные и текстурные заполнения с дополнительной прозрачностью для объектов пользовательского интерфейса.
- Анимированные раскадровки, которые могут использоваться в любых ситуациях, включая инициированные пользователем события, такие как щелчки на кнопках.
- Многократно используемые ресурсы, которые можно использовать для динамической стилизации элементов управления.

Большая часть этой функциональности специально ориентирована на операционную систему Microsoft Vista и последующие версии (Windows 7, Windows Server 2008 и Windows Server 2008 R2), которые обладают расширенными графическими возможностями, доступными через интерфейс Aero. Тем не менее, приложения WPF могут выполняться и под управлением других операционных систем, таких как Windows XP. Система нейтральной визуализации, встроенная в .NET Framework 4, умеет визуализировать XAML (с соответствующей потерей производительности), если графическая карта по каким-то причинам не в состоянии это делать.

Среды VS и VCE включают возможности создания и стилизации кода XAML, но наиболее подходящим инструментом для дизайнеров является Microsoft Expression Blend (EB). Это пакет для дизайна и компоновки, который можно использовать для проектов в любой среде с последующим редактированием в других средах. Все, что нужно в EB для редактирования файла отделенного кода — это выполнить двойной щелчок на нем в окне Files (Файлы) — эквиваленте окна Solution Explorer в VS и VCE. Интерфейс EB показан на рис. 25.2.

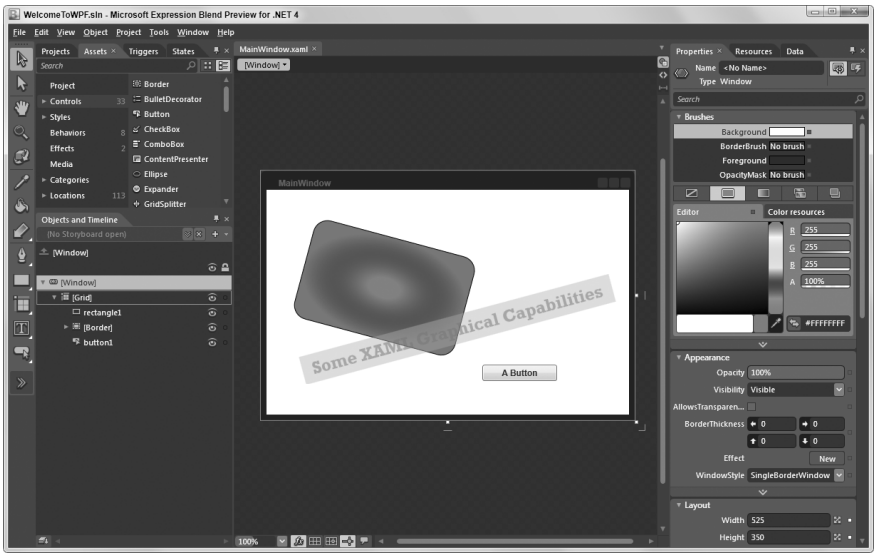


Рис. 25.2. *Интерфейс Microsoft Expression Blend*

Пробная версия EB доступна для загрузки по адресу `http://microsoft.com/expression/products/Blend_Overview.aspx`. Однако помните, что для написания приложений WPF или редактирования XAML среда EB не обязательна. На рис. 25.3 показан тот же проект, что и на рис. 25.3, но загруженный в VCE.

В VCE по умолчанию в двух панелях отображается XAML-разметка (пока не обращайте особого внимания на код, показанный в представлении XAML) и предварительный вид визуализированного XAML. Редактор свойств несколько менее интуитивно понятен.

Приложение может быть запущено из обеих сред с одинаковым результатом, показанным на рис. 25.4.

WPF для разработчиков на C#

Как было показано в предыдущем разделе, разработчики могут создавать проекты и решения, с которыми можно работать в VS и VCE, и *те же самые* проекты и решения могут редактироваться в Expression Blend. Однако в отличие от дизайнеров, разработчики тратят больше времени на работу в VS и VCE.

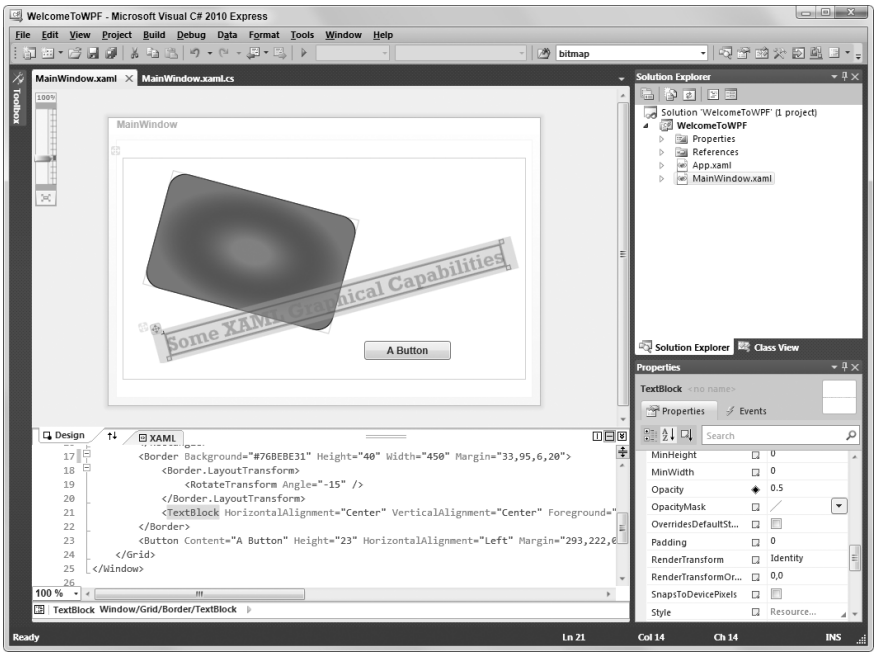


Рис. 25.3. *Среда VCE обладает рядом возможностей EB*

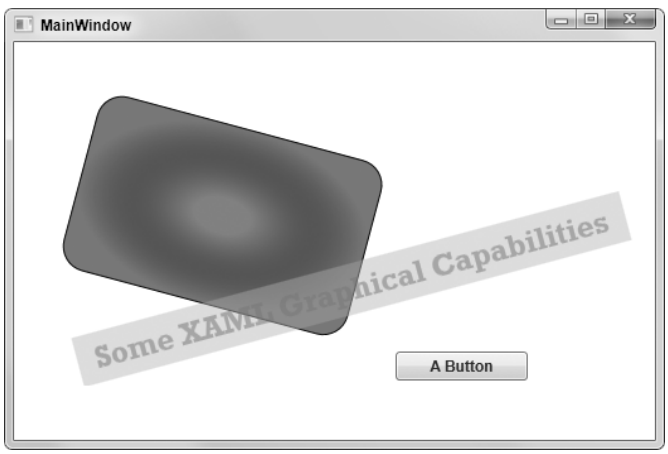


Рис. 25.4. *Результат запуска примера приложения*

В WPF, как вы узнали во введении к этому разделу, используется режим отделенного кода, подобный ASP.NET. Например, для создания обработчика событий в элементе управления `Button` необходимо добавить атрибут `Click` к XML-элементу, представляющему элемент управления. С помощью этого атрибута указывается имя обработчика событий в файле отделенного кода для страницы XAML, который можно написать на C#.

Обратите внимание на возможность манипулирования элементами управления в приложении WPF подобно тому, как в приложениях Windows Forms применяются программные

приемы для компоновки пользовательских интерфейсов. Отделенный код можно использовать для создания экземпляра элемента управления, установки его свойств, добавления обработчиков событий и вставки элемента управления в окно. Фактически это позволяет полностью обойтись без XAML. Необходимый код обычно длиннее, чем соответствующий декларативная разметка XAML, и, кроме того, утрачивается разделение между дизайном и функциональностью. Хотя в некоторых ситуациях приходится делать все это программно, обычно для компоновки элементов управления в пользовательском интерфейсе должен использоваться язык XAML.

В этой главе технология WPF представлена с точки зрения разработчика на C#. WPF — тема, о которой написаны целые книги, так что здесь мы ограничимся лишь кратким описанием доступных возможностей.

Структура базового приложения WPF

Технология WPF интуитивно понятна в использовании, и лучший способ изучить ее состоит в том, чтобы просто приступить к экспериментированию. Во время проработки оставшейся части книги многие приемы покажутся знакомыми.

В следующем практическом занятии будет создано приложение WPF, а в разделе "Описание работы" рассматривается код и его результаты, что позволит лучше понять, как все это работает вместе.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Создание базового приложения WPF

1. Создайте приложение WPF по имени `Ch25Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter25`.

2. Модифицируйте код в `MainWindow.xaml`, как показано ниже:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex01.MainWindow"
  Title="Color Spinner" Height="370" Width="270">
  <Window.Resources>
    <Storyboard x:Key="Spin">
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse1"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:10" Value="360"/>
      </DoubleAnimationUsingKeyFrames>
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse2"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:10" Value="-360"/>
      </DoubleAnimationUsingKeyFrames>
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse3"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:05" Value="360"/>
      </DoubleAnimationUsingKeyFrames>
```

```
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse4"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:05" Value="-360"/>
      </DoubleAnimationUsingKeyFrames>
    </Storyboard>
  </Window.Resources>
  <Window.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
      <BeginStoryboard Storyboard="{StaticResource Spin}"
        x:Name="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="goButton">
      <ResumeStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="stopButton">
      <PauseStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
    </EventTrigger>
  </Window.Triggers>
  <Window.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FFFFFFFF" Offset="0"/>
      <GradientStop Color="#FFFFC45A" Offset="1"/>
    </LinearGradientBrush>
  </Window.Background>
  <Grid>
    <Ellipse Margin="50,50,0,0" Name="ellipse5" Stroke="Black" Height="150"
      HorizontalAlignment="Left" VerticalAlignment="Top" Width="150">
      <Ellipse.Effect>
        <BlurEffect Radius="10"/>
      </Ellipse.Effect>
      <Ellipse.Fill>
        <RadialGradientBrush>
          <GradientStop Color="#FF000000" Offset="1"/>
          <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
        </RadialGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <Ellipse Margin="15,85,0,0" Name="ellipse1" Stroke="{x:Null}"
      Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
      Width="120" Fill="Red" Opacity="0.5"
      RenderTransformOrigin="0.92,0.5" >
      <Ellipse.Effect>
        <BlurEffect/>
      </Ellipse.Effect>
      <Ellipse.RenderTransform>
        <TransformGroup>
          <RotateTransform Angle="0"/>
        </TransformGroup>
      </Ellipse.RenderTransform>
    </Ellipse>
    <Ellipse Margin="85,15,0,0" Name="ellipse2" Stroke="{x:Null}"
      Height="120" HorizontalAlignment="Left" VerticalAlignment="Top"
      Width="80" Fill="Blue" Opacity="0.5"
      RenderTransformOrigin="0.5,0.92" >
      <Ellipse.Effect>
        <BlurEffect/>
      </Ellipse.Effect>
```

```
      <Ellipse.RenderTransform>
        <TransformGroup>
          <RotateTransform Angle="0"/>
        </TransformGroup>
      </Ellipse.RenderTransform>
    </Ellipse>
    <Ellipse Margin="115,85,0,0" Name="ellipse3" Stroke="{x:Null}"
      Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
      Width="120" Opacity="0.5" Fill="Yellow"
      RenderTransformOrigin="0.08,0.5" >
      <Ellipse.Effect>
        <BlurEffect/>
      </Ellipse.Effect>
      <Ellipse.RenderTransform>
        <TransformGroup>
          <RotateTransform Angle="0"/>
        </TransformGroup>
      </Ellipse.RenderTransform>
    </Ellipse>
    <Ellipse Margin="85,115,0,0" Name="ellipse4" Stroke="{x:Null}"
      Height="120" HorizontalAlignment="Left" VerticalAlignment="Top"
      Width="80" Opacity="0.5" Fill="Green"
      RenderTransformOrigin="0.5,0.08" >
      <Ellipse.Effect>
        <BlurEffect/>
      </Ellipse.Effect>
      <Ellipse.RenderTransform>
        <TransformGroup>
          <RotateTransform Angle="0"/>
        </TransformGroup>
      </Ellipse.RenderTransform>
    </Ellipse>
  <Button Height="23" HorizontalAlignment="Left" Margin="20,0,0,56"
    Name="goButton" VerticalAlignment="Bottom" Width="75" Content="Go"/>
  <Button Height="23" HorizontalAlignment="Left" Margin="152,0,0,56"
    Name="stopButton" VerticalAlignment="Bottom" Width="75" Content="Stop"/>
  <Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
    Name="toggleButton" VerticalAlignment="Bottom" Width="75" Content="Toggle"/>
  </Grid>
</Window>
```

Фрагмент кода ***Ch25Ex01\MainWindow.xaml***

3. Дважды щелкните на кнопке Toggle (Переключить) в представлении дизайна (на рис. 25.5 она выделена, а представление XAML свернуто).
4. Модифицируйте код в `MainWindow.xaml.cs`, как показано ниже (обратите внимание на оператор `using` и обработчик событий `toggleButton_Click()`, добавленный в результате двойного щелчка на кнопке Toggle).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
```

```
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Media.Animation;
namespace Ch25Ex01
{
  /// <summary>
  /// Логика взаимодействия для MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }
    private void toggleButton_Click(object sender, RoutedEventArgs e)
    {
      Storyboard spinStoryboard = Resources["Spin"] as Storyboard;
      if (spinStoryboard != null)
      {
        if (spinStoryboard.GetIsPaused(this))
        {
          spinStoryboard.Resume(this);
        }
        else
        {
          spinStoryboard.Pause(this);
        }
      }
    }
  }
}
```

Фрагмент кода ***Ch25Ex01\MainWindow.xaml.cs***

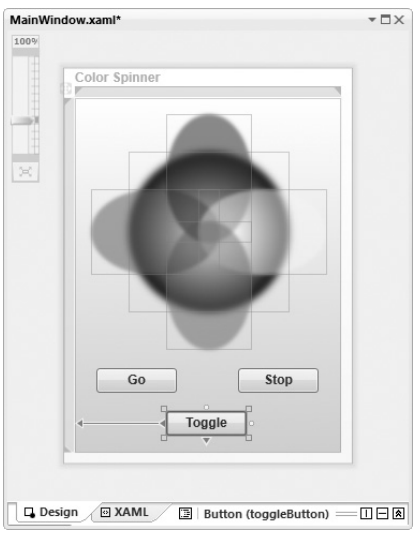


Рис. 25.5. *Выбор кнопки для добавления обработчика события* `Click`

Рис. 25.6. *WPF-приложение* `Ch25Ex01` *в работе*

5. Запустите приложение и поэкспериментируйте со стартом, остановкой и переключением анимации. Пример показан на рис. 25.6.
6. Создайте новое приложение типа WPF Browser Application (Браузерное WPF-приложение) по имени `Ch25Ex01Web` и сохраните его в каталоге `C:\BegVCSharp\Chapter25`.
7. Измените значение атрибута `Title` элемента `<Page>` в `Page1.xaml` на `Color Spinner Web`.
8. Откройте файл `MainWindow.xaml` приложения `Ch25Ex01` и скопируйте весь код элемента `<Window>` из этого файла в элемент `<Page>` в файле `Page1.xaml`.
9. Замените элементы `<Window.Resources>`, `<Window.Triggers>` и `<Window.Background>` соответственно на `<Page.Resources>`, `<Page.Triggers>` и `<Page.Background>` (не забудьте изменить как открывающий, так и закрывающий дескрипторы элементов).
10. Удалите элемент `<Ellipse.Effect>` вместе с его содержимым из `Page1.xaml`.
11. Скопируйте обработчик событий `toggleButton_Click()` и оператор `using` для пространства имен `System.Windows.Media.Animation` из `MainWindow.xaml.cs` в `Page1.xaml.cs`.
12. Запустите приложение `Ch25Ex01Web`. Результат можно видеть на рис. 25.7.

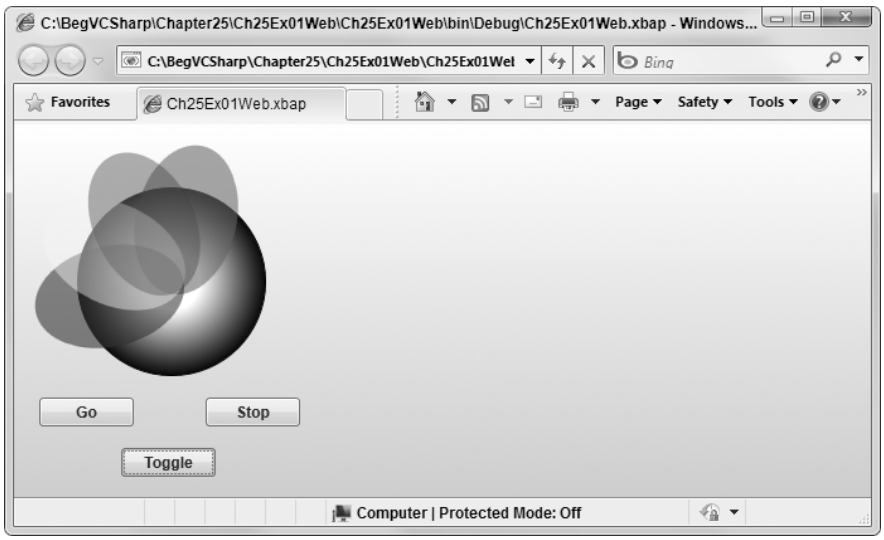


Рис. 25.7. *Веб-приложение* `Ch25Ex01Web` *в работе*

Описание работы

В этом примере было создано простое приложение, которое выводит вращающиеся цветные эллипсы, которые можно запускать или останавливать. К сожалению, черно-белые снимки экрана не могут продемонстрировать эффект в полной мере, но если вы запустите код самостоятельно, то обнаружите, что приложение выглядит эстетически привлекательно.

Для достижения этого результата мы добавили изрядный объем XAML, но если вы присмотритесь к нему, то заметите, что в нем много повторяющегося кода, потому что аними-

руются четыре эллипса. Возможно, вы также заметите, что в файл отделенного кода было добавлено совсем немного кода, и этот код предназначен только для одной из трех кнопок. Код спроектирован так, чтобы проиллюстрировать два важных момента.

- Дизайнеры могут создавать привлекательные пользовательские интерфейсы, включающие расширенные графические возможности и средства взаимодействия с пользователем, обходясь исключительно XAML-разметкой.
- При необходимости можно получить полный контроль над пользовательским интерфейсом, определенным с помощью XAML, из отделенного кода.

Здесь также было показано, как использовать в веб-приложении тот же самый код, что и в настольном приложении. Понадобилось совсем немного модификаций, которые вы увидите позже, но базовая функциональность одинакова в обеих средах.

В коде этого приложения демонстрируются многие ключевые средства WPF. Для начала рассмотрим настольное приложение, а затем — изменения, необходимые для веб-приложения. Итак, первым делом взгляните на элемент верхнего уровня в XAML-разметке для настольного приложения, `MainWindow.xaml`:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex01.MainWindow"
  Title="Color Spinner" Height="370" Width="270">
  ...
</Window>
```

Элемент `<Window>` используется для определения окна. Приложение может состоять из нескольких окон, каждое из которых описано в отдельном файле XAML. Однако это не значит, что файл XAML всегда определяет окно; файл XAML может создавать пользовательские элементы управления, кисти и прочие ресурсы, веб-страницы и т.п. В проекте `Ch25Ex01` имеется даже файл XAML, определяющий само приложение — `App.xaml`. Чуть позже в этой главе мы рассмотрим приложение и файл `App.xaml`.

Возвращаясь к `MainWindow.xaml`, обратите внимание, что элемент `<Window>` содержит некоторые очевидные атрибуты. Есть два объявления пространств имен: одно для глобального пространства имен, используемого для XML, и одно — для пространства имен `x`. Затем идет атрибут `Class` из пространства имен `x`. Этот атрибут связывает XAML-элемент `<Window>` с определением частичного класса в отделенном коде, в данном случае — `Ch25Ex01.Window`. Это похоже на то, как работает ASP.NET, с классом, используемым для страницы, и позволяет отделенному коду совместно использовать ту же кодовую модель, что и файл XAML, включая элементы управления, определенные элементами XAML, и т.д. Следует отметить, что атрибут `x:Class` может использоваться только в корневом элементе файла XAML.

Три других атрибута — `Title`, `Height` и `Width` — задают текст для отображения в заголовке окна, а также размеры (в пикселях) окна. Эти атрибуты отображаются на свойства класса `System.Windows.Window`, от которого унаследован класс `Ch25Ex01.Window`.

Несколько других свойств класса `System.Windows.Window` позволяют определять дополнительную функциональность. Многие из этих свойств более сложны, чем те три, что применяются в элементе `<Window>`, т.е. они, например, не являются простыми строками или числами. Синтаксис XAML позволяет использовать вложенные элементы для указания значений этих свойств.

Синтаксис, использованный в XAML для определения объектов, свойств и содержимого, более детально обсуждается в разделе “Синтаксис XAML”.

Например, свойство `Background` определено в этом коде с вложенным элементом `<Window.Backgrond>`:

```
<Window.Background>
  <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="#FFFFFFFF" Offset="0"/>
    <GradientStop Color="#FFFFC45A" Offset="1"/>
  </LinearGradientBrush>
</Window.Background>
```

Этот код устанавливает свойство Background в экземпляр класса LinearGradientBrush. В данном случае кисть определяет градиент, который изменяется от белого до персикового — сверху вниз.

Есть два других "сложных" свойств, определенных во вложенных элементах в этом коде: <Window.Resources>, определяющее анимацию, и <Window.Triggers>, которое определяет триггеры, управляющие анимацией. Оба эти свойства — Resources и Triggers — способны на гораздо большее, и далее в этой главе они рассматриваются более подробно.

Перед тем, как рассмотреть реализацию этих свойств, стоит немного забежать вперед и упомянуть элемент <Grid>. Элемент <Grid> определяет экземпляр элемента управления System.Windows.Controls.Grid. Это один из тех элементов управления, которые можно использовать для компоновки в приложениях WPF. Он позволяет позиционировать вложенные элементы управления, используя координаты относительно любой из граней прямоугольника. Другие элементы управления дают возможность позиционировать элементы управления другими способами. Все элементы управления компоновкой описаны далее в этой главе, в разделе "Управление компоновкой".

Элемент <Grid> содержит пять элементов <Ellipse> (System.Windows.Shapes.Ellipse). Эти элементы определяют эллипсы, используемые для отображения вращающейся графики в приложениях, и кнопки, применяемые для управления приложением.

Первый элемент <Ellipse> определен так:

```
<Ellipse Margin="50,50,0,0" Name="ellipse5" Stroke="Black" Height="150"
  HorizontalAlignment="Left" VerticalAlignment="Top" Width="150">
  <Ellipse.Effect>
    <BlurEffect Radius="10"/>
  </Ellipse.Effect>
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF000000" Offset="1"/>
      <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

В этом элементе определен экземпляр класса System.Windows.Shapes.Ellipse, который применяется для отображения фигуры — эллипса, и устанавливает несколько свойств этого элемента управления следующим образом.

- Name. Идентификатор элемента управления.
- Margin. Указывает расположение фигуры, определенной элементом Ellipse в содержащей его сетке, задавая поля, окружающие фигуру. Размер полей задается в пикселях в коде. Как это свойство отображается на действительное местоположение фигуры, зависит от свойств HorizontalAlignment и VerticalAlignment.
- HorizontalAlignment и VerticalAlignment. Используются для спецификации граней прямоугольника, определенного Grid, которые служат для компоновки фигуры. Например, значения Left и Bottom заставляют фигуру позиционироваться относительно нижнего левого угла Grid.
- Height и Width. Размеры фигуры.

- `Stroke`. Кисть, которая используется для очерчивания границ фигуры, определенной элементом `Ellipse`.
- `Fill`. Кисть, которая используется для заполнения внутренней части фигуры, определенной элементом `Ellipse`.
- `Effect`. Специальный эффект, используемый при отображении элемента управления `Ellipse`.

Для свойства `Fill` применяется кисть `RadialGradientBrtush`. В данном случае кисть определяет градиент от белого (в центре эллипса) до черного (у его границ) цвета.

Свойство `Effect` установлено в `BlurEffect`. Это один из двух эффектов по умолчанию, которые можно применять к графике WPF. Он размывает фигуру в степени, заданной свойством `BlurEffect.Radius`. Этот эффект недоступен в веб-приложениях, и потому он был удален на шаге 10. Это одно из отличий между настольными и веб-приложениями.

Имеется возможность определить собственные эффекты для применения к элементам XAML, но эта сложная технология в настоящей главе не рассматривается.

Еще четыре элемента `<Ellipse>` в коде очень похожи. Каждый из них определяет один из анимированных цветных эллипсов. Ниже показан первый из них:

```
<Ellipse Margin="15,85,0,0" Name="ellipse1" Stroke="{x:Null}"
  Height="80" HorizontalAlignment="Left" VerticalAlignment="Top"
  Width="120" Fill="Red" Opacity="0.5"
  RenderTransformOrigin="0.92,0.5" >
  <Ellipse.Effect>
    <BlurEffect/>
  </Ellipse.Effect>
  <Ellipse.RenderTransform>
    <TransformGroup>
      <RotateTransform Angle="0"/>
    </TransformGroup>
  </Ellipse.RenderTransform>
</Ellipse>
```

Этот код выглядит очень похожим на код предыдущего эллипса, но здесь имеются следующие отличия.

- Свойство `Stroke` установлено в `{x:null}`. В XAML значения, заключенные в фигурные скобки вроде этих, называются расширениями разметки и используются для указания значений свойств, которые не могут быть выражены простыми строками в синтаксисе XAML. В данном случае `{x:null}` указывает значение `null` для свойства, означающее, что кисть для `Stroke` не используется.
- Свойство `Opacity` специфицировано со значением `0.5`. Это задает полупрозрачность эллипса.
- Свойство `Effect` использует `BlurEffect` без атрибута `Radius`; в этом случае `Radius` получает значение по умолчанию, равное `5`.
- Свойство `RenderTRansform` установлено в объект `TransformGroup` с единственной трансформацией — `RotateTransform`. Эта трансформация используется при анимации эллипса. У него указано единственное свойство — `Angle`, которое задает угол поворота эллипса в градусах и изначально установлено в `0`.
- Свойство `RenderTransformOrigin` используется для установки центральной точки, вокруг которой будет поворачиваться эллипс в соответствии с информацией `RotateTransform`.

Два последних свойства касаются анимации, определенной в XAML, которая задается объектом `System.Windows.Media.Animation.Storyboard`. Этот объект определен в элементе `<Window.Resources>`, а это значит, что объект `Storyboard` будет доступен через коллекцию `Resources` окна. Код также определяет атрибут `x:Key`, который позволяет ссылаться на объект `Storyboard` через `Resources` посредством ключа:

```
<Window.Resources>
  <Storyboard x:Key="Spin">
    ...
  </Storyboard>
</Window.Resources>
```

Объект `Storyboard` содержит четыре объекта `DoubleAnimationUsingKeyFrames`. Эти объекты позволяют указать, что свойство, содержащее значение `double`, должно изменяться со временем, наряду с деталями, дополнительно определяющими это поведение. Каждый из элементов в этом коде определяет анимацию, используемую одним из цветных эллипсов. Например, анимация рассмотренного ранее эллипса `ellipse1` выглядит следующим образом:

```
      <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
        Storyboard.TargetName="ellipse1"
        Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0].(RotateTransform.Angle)"
        RepeatBehavior="Forever">
        <SplineDoubleKeyFrame KeyTime="00:00:10" Value="360"/>
      </DoubleAnimationUsingKeyFrames>
```

Пока не слишком вдаваясь в подробности этого элемента, отметим, что он указывает, что свойство `Angle` описанной ранее трансформации `RotateTransform` должно изменяться от начального значения до значения 360 за период времени в 10 секунд, и что это изменение должно повторяться после завершения. В разделе "Анимация" настоящей главы разновидности анимации рассматриваются более подробно.

После определения эллипса находятся три элемента `<Button>`, определяющие кнопки (обратите внимание, что атрибут `Click` в коде настоящего примера не показан; его добавляет IDE-среда, когда вы дважды щелкаете на кнопке на шаге 3):

```
<Button Height="23" HorizontalAlignment="Left" Margin="20,0,0,56"
  Name="goButton" VerticalAlignment="Bottom" Width="75" Content="Go"/>
<Button Height="23" HorizontalAlignment="Left" Margin="152,0,0,56"
  Name="stopButton" VerticalAlignment="Bottom" Width="75" Content="Stop"/>
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggle" Click="toggleButton_Click"/>
```

Каждый из этих элементов задает имя, позицию и размеры объекта `Button`, используя те же свойства, что и элементы `<Ellipse>`, показанные ранее. Они также имеют свойства `Content`, определяющие то, что отображается в содержимом кнопки — в данном случае отображается строка текста надписи на кнопке. Кнопки могут отображать подобным образом не только простые строки; при желании можно использовать встроенные фигуры или графическое содержимое. Подробности изложены далее, в разделе "Стилизация элементов управления".

Атрибут `Click` кнопки `toggleButton` определяет метод — обработчик события `Click`. Этот метод, `toggleButton_Click()`, на самом деле является обработчиком *маршрутизируемого события*. Эти события рассматриваются в разделе "Маршрутизируемые события" далее в главе. Пока достаточно знать, что это событие инициируется, когда совершается щелчок на кнопке; в этом случае вызывается обработчик данного события.

В коде обработчика события сначала получается ссылка на объект `Storyboard`, определяющий анимацию. Ранее было показано, что этот объект содержится в свойстве

Resiurces включающего объекта Window, и что он использует ключ Spin. Код, который извлекает Storyboard, не должен вызвать никаких вопросов:

```
private void toggleButton_Click(object sender, RoutedEventArgs e)
{
    Storyboard spinStoryboard = Resources["Spin"] as Storyboard;
```

Если значение не равно null, то с помощью метода Storyboard.GetIsPaused() определяется, приостановлена ли в данный момент анимация. Если да, выполняется вызов Resume(), если же нет — вызывается Pause(). Эти методы, соответственно, возобновляют или приостанавливают анимацию.

```
  if (spinStoryboard != null)
  {
    if (spinStoryboard.GetIsPaused(this))
    {
      spinStoryboard.Resume(this);
    }
    else
    {
      spinStoryboard.Pause(this);
    }
  }
}
```

Обратите внимание, что все эти методы принимают ссылку на объект, содержащий раскадровку (storyboard). Причина в том, что раскадровки сами по себе не отслеживают время. Окно, содержащее раскадровку, имеет свои собственные часы, используемые ею. Получая ссылку на окно (с помощью this), раскадровка может получить доступ к часам.

Другие две кнопки — goButton и stopButton — не привязаны ни к какому обработчику событий в отделенном коде. Вместо этого их функциональность задается триггерами. В данном примере определены три триггера, как показано ниже:

```
<Window.Triggers>
  <EventTrigger RoutedEvent="FrameworkElement.Loaded">
    <BeginStoryboard Storyboard="{StaticResource Spin}"
      x:Name="Spin_BeginStoryboard"/>
  </EventTrigger>
  <EventTrigger RoutedEvent="ButtonBase.Click"
    SourceName="goButton">
    <ResumeStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
  </EventTrigger>
  <EventTrigger RoutedEvent="ButtonBase.Click"
    SourceName="stopButton">
    <PauseStoryboard BeginStoryboardName="Spin_BeginStoryboard"/>
  </EventTrigger>
</Window.Triggers>
```

Первый из них — триггер, связывающий событие FrameworkElement.Loaded (которое инициируется при загрузке приложения) с действием BeginStoryboard. Это действие запускает анимацию Spin. Ссылка на анимацию Spin осуществляется с применением синтаксиса расширения разметки — кодом {StaticResource.Spin}. Этот синтаксис, используемый для обращения к ресурсам во включающем окне, часто применяется в приложениях WPF. Действие BeginStoryboard получает имя Spin_Storyboard и ссылается в других двух триггерах, которые привязывают события Click кнопок goButton и stopButton, соответственно. Эти триггеры используют действия ResumeStoryboard и PauseStoryboard, которые возобновляют выполнение и останавливают раскадровку.

Приведенный код хорошо работает в настольном приложении, но при преобразовании в веб-приложение в него необходимо внести несколько изменений. Фактически в

примере некоторые из этих деталей скрываются за счет создания браузерного веб-приложения (WPF Browser). Например, поскольку определенные ограничения безопасности ассоциируются с запуском кода в браузере, приложение WPF Browser Application оснащено временным ключом, который служит для подписи приложения. Это необходимо, если приложение должно выполнять действия, которые запрещены в браузерном приложении, например, обращение к локальной файловой системе.

Возможно, вы также заметили, что корневой элемент `<Window>` настольного приложения в веб-приложении заменен элементом `<Page>`. Это связано с тем, что средства, предоставляемые браузером, слегка отличаются от средств, предлагаемых хост-приложением, которое используется для запуска настольных приложений. Поэтому для представления таких разных хостов в WPF применяются разные классы. Однако, как показано в коде, для многих вещей можно использовать идентичный код в той же самой среде. Далее в главе мы рассмотрим это более подробно.

На этом анализ примера приложения завершен. Вы получили хорошую основу и ознакомились с множеством новых концепций. Остаток главы будет посвящен углубленному рассмотрению описанных приемов, формализации требуемого синтаксиса и представлению нескольких новых вещей.

Основы WPF

Пример, приведенный в начале этой главы, должен был вселить энтузиазм относительно программирования с помощью WPF. Хотя пришлось охватить множество новых концепций, вы увидели, что комбинация XAML и кода .NET позволяет очень быстро создавать динамические приложения. Также было показано, что при желании можно передать массу функциональности, включая пользовательский интерфейс приложения, в руки дизайнеров, от которых не требуется знание C#. Наконец, вы увидели, как создается настольное и веб-приложение с (более или менее) одинаковым кодом.

Однако, прежде чем приступать к самостоятельному созданию приложения WPF, стоит посвятить некоторое время изучению основ. В этом разделе будет рассмотрен ряд тем, основополагающих для приложений WPF, а также показан синтаксис, необходимый для их реализации. Кроме того, вы узнаете о множестве дополнительных возможностей, которые впоследствии исследуете в собственных приложениях.

Итак, в настоящем разделе будут охвачены следующие темы:

- синтаксис XAML;
- настольные и веб-приложения;
- объект `Application`;
- основы элементов управления, включая свойства зависимости, присоединенные свойства, маршрутизируемые события и присоединенные события;
- компоновка и стилизация элементов управления;
- триггеры;
- анимация;
- статические и динамические ресурсы.

Синтаксис XAML

В примере, приведенном в начале главы, была продемонстрирована значительная часть синтаксиса без формального описания. Многие из этих правил и возможностей были опущены, чтобы можно было сосредоточиться на общей структуре и функциональности. В настоящем разделе мы рассмотрим XAML чуть более подробно, чтобы показать, из чего состоят файлы XAML.

Синтаксис объектного элемента

Базовая структура файла XAML использует *синтаксис объектного элемента* (object element syntax) для описания иерархии объектов с единым корневым объектом, содержащим все остальное. Синтаксис объектного элемента описывает объект (или структуру), представленный элементом XML. Например, вы видели в примере, как элемент <Button> использовался для представления объекта System.Windows.Controls.Button.

В корневом элементе файла XAML всегда применяется синтаксис объектного элемента, хотя, как было показано в приведенном ранее примере, класс, используемый для корневого объекта, определяется не именем элемента (<Window> или <Page>), а атрибутом x:Class. Этот синтаксис применяется только в корневом элементе. Для настольных приложений корневой элемент должен быть унаследован от System.Windows.Window, а для веб-приложений — от System.Windows.Controls.Page.

Многие из объектов, которые определяются с использованием синтаксиса объектного элемента, фактически являются элементами управления, такими как элемент Button, использованный в данном примере.

Синтаксис атрибутов

Во многих случаях, когда элемент используется для представления объекта (с помощью синтаксиса объектного элемента), атрибуты служат для указания свойств и событий. Например, в показанном ранее элементе <Button> атрибуты применяются следующим образом:

```
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggle" Click="toggleButton_Click"/>
```

Здесь каждый атрибут устанавливает значение свойства объекта toggleButton, кроме Click, который назначает обработчик маршрутизируемого события Click элемента toggleButton, и Name. Все это — примеры *синтаксиса атрибутов*.

Атрибут Name здесь представляет собой специальный случай: он определяет идентификатор для элемента управления, необходимый для того, чтобы можно было ссылаться на него из отделенного кода и другого кода XAML.

Атрибуты могут быть квалифицированы базовым классом, на который они ссылаются, указанным через точку. Например, элемент управления Button наследует свое событие Click от ButtonBase, поэтому предыдущий код можно переписать следующим образом:

```
<Button Height="23" HorizontalAlignment="Left" Margin="85,0,86,16"
  Name="toggleButton" VerticalAlignment="Bottom" Width="75"
  Content="Toggle" ButtonBase.Click="toggleButton_Click"/>
```

Обратите внимание, что этот же синтаксис также используется для *присоединенных свойств* (attached properties), которые будут рассматриваться в разделе “Основы элементов управления”.

Синтаксис элемента свойства

Во многих случаях для инициализации значения свойства должно использоваться нечто более сложное, чем простая строка. В примере приложения это делалось для свойств Fill, которые устанавливались в различные объекты кистей:

```
<Ellipse ...>
  ...
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF000000" Offset="1"/>
      <GradientStop Color="#FFFFFFFF" Offset="0.306"/>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Здесь свойство устанавливается дочерним элементом, который назван в соответствии со следующим соглашением:

```
[Имя_родительского_элемента].[Имя_свойства]
```

Это и называется *синтаксисом элемента свойства*.

Синтаксис содержимого

Многие элементы управления на самом деле являются *презентаторами содержимого* (content presenter). Это значит, что элемент управления снабжается содержимым, отображаемым в соответствии с его шаблоном. Например, содержимое, указанное для элемента управления `Button`, отображается на поверхности кнопки. Это может быть текст, как в примере, или же какая-то графика.

С помощью *синтаксиса содержимого* содержимое элемента управления указывается просто путем добавления его в элемент, представляющий элемент управления:

```
<Button ...>Go</Button>
```

Это код заставит элемент управления `Button` отобразить текст `Go` на своей поверхности во время визуализации. В примере используется более простой, но менее гибкий способ сделать это — с помощью атрибута по имени `Content`. Применяемый здесь полный синтаксис содержимого необходим для отображения более сложного графического содержимого, описанного во введении к этой главе.

В результате использования сложного содержимого для элемента управления получается сложная вложенная иерархия кода XAML. По этой причине XAML позволяет стилизовать элементы управления другими, более подходящими средствами. Подробнее о презентаторах содержимого и стилизации читайте в разделе “Стилизация элементов управления”.

Смешивание синтаксиса элемента свойства и синтаксиса содержимого

Элемент на странице XAML, сформатированный в синтаксисе объектного элемента, может включать и синтаксис свойств элемента, и синтаксис содержимого. Если это так, необходимо придерживаться следующих синтаксических правил.

- Элементы, используемые для синтаксиса элемента свойства, не должны быть непрерывными, т.е. можно иметь элемент, использующий синтаксис элемента свойства, за которым идет элемент, в котором применяется синтаксис содержимого, а за ним — элемент, использующий синтаксис элемента свойства.
- Элементы (и текстовое содержимое, если оно есть), используемые для синтаксиса содержимого, должны быть непрерывными, т.е. нельзя иметь текст или элемент, использующий синтаксис содержимого, за которым следует элемент с синтаксисом элемента свойства, а за ним — элемент, снова использующий синтаксис содержимого.

Таким образом, показанный ниже код является допустимым:

```
<Button ...>
  <Button.Effect>
    <BlurEffect Radius="10"/>
  </Button.Effect>
  Go
  <Button.RenderTransform>
    <RotateTransform Angle="20"/>
  </Button.RenderTransform>
</Button>
```

Однако следующий фрагмент кода работать не будет, потому что два его места, где используется синтаксис содержимого, разделены элементом с синтаксисом элемента свойства:

```
<Button ...>
  Don't
  <Button.Effect>
    <BlurEffect Radius="10"/>
  </Button.Effect>
  Go
</Button>
```

Расширения разметки

В примере уже демонстрировалось, что для значений свойств могут использоваться расширения разметки, например, `{x:null}`. Везде, где встречаются фигурные скобки, применяются *расширения разметки* (markup extensions). Они могут использоваться как в коде синтаксиса атрибутов, так и в коде синтаксиса элемента свойства.

Все расширения разметки в настоящей главе являются специфичными для WPF. Они также включают расширения, служащие для ссылки на ресурсы и для привязки данных.

Настольные и веб-приложения

В примере, приведенном ранее в этой главе, было продемонстрировано, что приложение WPF может быть запущено и как автономное настольное приложение, и как веб-приложение. В качестве начальной точки применялись шаблоны проектов WPF Application и WPF Browser Application, а для завершения приложения добавлялся код XAML и C#. Шаблон WPF Application обеспечивает компиляцию проекта в файл `.exe`, а шаблон WPF Browser Application — в файл `.xbap`.

XBAP — это сокращение от XAML Browser Application (Браузерное приложение XAML), и веб-приложения, использующие WPF, часто называют XBAP-приложениями.

Большая часть различий между этими двумя типами приложений касается файлов проектов (`.csproj`). Приложения WPF Browser Application имеют несколько дополнительных настроек, включая установку подписи как для приложения, так и для манифеста приложения (из упомянутых ранее соображений безопасности), а также для разрешения отладки в браузере. Как упоминалось ранее, существует тестовый сертификат, который можно использовать для такой подписи. В рабочей среде этот сертификат должен быть заменен действительным сертификатом, полученным от центра сертификации (certification authority).

Преобразование между приложениями WPF и Web WPF может быть пустяковой задачей, поскольку при этом требуется изменить относительно немного настроек, и (как было показано в примере) внести определенные изменения в код XAML. К ним относятся замена элементов `<Window>` элементами `<Page>` и удаление такой функциональности, как растровые эффекты. Наилучший подход обычно предусматривает создание отдельных проектов, как это делалось в предыдущем примере.

Объект `Application`

В WPF большинство приложений (включая XBAP-приложения и настольные приложения, использующие шаблон WPF Application) содержат экземпляр класса, унаследованного от `System.Windows.Application`. В показанном ранее примере приложения этот объект определен в файлах `App.xaml` и `App.xaml.cs`. Содержимое `App.xaml` для `Ch25Ex01` выглядит следующим образом:

```
<Application x:Class="Ch25Ex01.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
```

```
  <Application.Resources>
  </Application.Resources>
</Application>
```

Фрагмент кода ***Ch25Ex01\App.xaml***

Синтаксис элемента `<Application>` подобен рассмотренному ранее синтаксису `<Window>`, и он аналогичным образом использует атрибут `x:Class`, чтобы связать код XAML с определением частичного класса в отделенном коде. Объект, определенный этим кодом — это входная точка приложения WPF. Может существовать только один экземпляр этого объекта, доступный в коде через статическое свойство `Application.Current`. Объект приложения исключительно удобен по перечисленным ниже причинам.

- Он предоставляет многочисленные события, инициируемые в определенных точках времени жизни приложения. К ним относится показанное ранее событие `LoadCompleted`, которое инициируется, когда приложение загружено и визуализировано, событие `DispatcherUnhandledException`, инициируемое при генерации необработанного исключения, и многие другие.
- Он содержит методы, которые можно использовать для установки и загрузки cookie-наборов, нахождения и загрузки ресурсов, и многого другого.
- Он содержит несколько свойств, которые можно использовать для доступа, например, к ресурсам контекста приложения (см. раздел "Статические и динамические ресурсы") и окнам приложения.

Пожалуй, наиболее полезными в этом списке являются события, инициированные объектом приложения, которые, скорее всего, будут использоваться наиболее часто.

Основы элементов управления

Технология WPF снабжает большим набором элементов, которые можно использовать при создании приложений. В этой главе приведен исчерпывающий обзор WPF и ее возможностей, что позволит вместо детального исследования каждого элемента управления посмотреть на них в действии.

Ниже приведен список элементов управления, предлагаемых в WPF:

```
Border                    Image                 Slider
BulletDecorator           Label                 StatusBar
Button                    ListBox               TabControl
Calendar                  ListView              TextBlock
CheckBox                  ListView              TextBox
ComboBox                  Menu                  ToolBar
ContextMenu               MediaElement          ToolTip
DataGrid                  PasswordBox           TreeView
DatePicker                Popup                 ViewBox
DocumentViewer            ProgressBar
Expander                  RadioButton
FlowDocumentPageViewer    RepeatButton
FlowDocumentReader        RichTextBox
FlowDocumentScrollViewer  ScrollBar
Frame                     ScrollViewer
GroupBox                  Separator
```

Список элементов управления не включает элементы WPF, используемые для компоновки, которые описаны далее в этой главе.

Некоторые из перечисленных элементов имеют очень похожие имена, и на самом деле их функциональность очень похожа на предлагаемую аналогичными элементами в приложениях Windows Forms и ASP.NET. Например, вы уже видели, как использовать элемент управления `Button` для визуализации кнопки. Другие элементы менее схожи, и с ними стоит поэкспериментировать, чтобы посмотреть, на что они способны.

Эти элементы управления изначально имеют базовый вид. Чтобы получить нечто более впечатляющее, их следует стилизовать, и именно здесь становится очевидной реальная мощь WPF, в чем вы вскоре убедитесь. Помимо стилизации есть несколько других средств, используемых элементами управления WPF. В этом разделе рассматриваются следующие возможности:

- свойства зависимости;
- присоединенные свойства;
- маршрутизируемые события.

Как и в других системах разработки настольных и веб-приложений, вы можете (и почти наверняка будете) создавать собственные элементы управления. При создании элемента управления можно использовать все описанные здесь средства. В последующих разделах представлены примеры реализации.

Свойства зависимости

Свойство зависимости (dependency property) — это разновидность свойств, используемых в WPF повсеместно, в частности, в элементах управления, которые предоставляют функциональность, расширяющую обычные свойства .NET. Для иллюстрации этого рассмотрим обычное свойство .NET. При создании классов в .NET, как правило, реализуются свойства с помощью очень простого кода вроде показанного ниже:

```
private string aStringProperty;
public string AStringProperty
{
  get
  {
    return aStringProperty;
  }
  set
  {
    aStringProperty = value;
  }
}
```

Здесь мы имеем общедоступное свойство, которое поддерживается приватным полем. Эта простая реализация подходит для большинства целей, но не предлагает сложной функциональности помимо базового доступа к состоянию. Например, если нужно добавить к элементу управления (`ControlA`) свойство `AStringProperty`, и другой элемент управления (`ControlB`) должен реагировать на изменения свойства, придется выполнить следующие шаги.

1. Добавьте к элементу `ControlA` свойство `AStringProperty` с использованием показанного ранее кода.
2. Добавьте к элементу `ControlA` событие.
3. Добавьте к элементу `ControlA` метод, инициирующий событие.
4. Добавьте код в метод доступа `set` свойства `AStringProperty` в элементе `ControlA` для вызова метода, инициирующего событие.
5. Добавьте к элементу `ControlB` код для подписки на событие, представленное `ControlA`.

Углубляться в код, необходимый для добавления и реакции на изменения простого свойства, нет необходимости, поскольку он уже неоднократно демонстрировался в этой книге.

С этим подходом связана одна проблема: отсутствие определенных стандартов, которым можно следовать для достижения результата. Разные разработчики могут добавить код, достигающий одинакового результата различными способами. Более того, это потребует идентифицировать все свойства, которые нуждаются в уведомлениях, во время разработки элемента управления.

Подход WPF к этой проблеме состоит в замене простого определения свойства, использованного в предыдущем коде, свойством зависимости с последующим применением формализованных, структурированных приемов для реализации уведомлений о его изменении. Свойством зависимости называется такое свойство, которое регистрируется системой свойств WPF так, чтобы обеспечить расширенную функциональность. Эта расширенная функциональность включает (но не ограничивается) автоматические уведомления об изменениях. В частности, свойства зависимости обладают следующими средствами.

- Для изменения значений свойств зависимости можно использовать стили.
- Устанавливать значение свойства зависимости можно с использованием ресурсов или привязки данных.
- Значения свойств зависимости можно изменять в анимации.
- Свойства анимации можно устанавливать иерархически в XAML, т.е. значение свойства зависимости, которое устанавливается в родительском элементе, может быть использовано для установки значения по умолчанию того же свойства зависимости дочерних элементов.
- Можно сконфигурировать уведомления для изменений значения свойства, используя хорошо определенный шаблон кодирования.
- Можно сконфигурировать наборы взаимосвязанных свойств, чтобы они обновлялись в ответ на изменения одного из них. Это называется *принуждением* (coercion). Говорят, что изменяющееся свойство *принуждает* изменять значения других свойств.
- К свойству зависимости можно применять метаданные, задавая другие поведенческие характеристики. Например, можно указать, что в случае изменения определенного свойства может понадобиться реорганизовать пользовательский интерфейс.

На практике из-за способа реализации свойств зависимости поначалу можно не заметить их существенного отличия от обычных свойств. Однако, создавая собственные элементы управления, вы быстро обнаружите, что при использовании обычных свойств .NET масса функциональности неожиданно становится недоступной.

Ввиду важности свойств зависимости для WPF позднее в этой главе будет показано, как их реализовать.

Присоединенные свойства

Присоединенное свойство (attached property) — это свойство, которое доступно каждому дочернему объекту экземпляра класса, определяющего свойство. Например, предположим, что имеется класс по имени `Recipe` (рецепт), который содержит дочерние объекты, представляющие ингредиенты. В определении класса `Recipe` можно определить присоединенное свойство по имени `Quantity`, которое затем будет использоваться всеми дочерними объектами. Обратите внимание, что дочерние объекты не обязаны задавать значения для присоединенных свойств.

Основная причина делать это состоит в том, что код XAML, который применяется для установки присоединенных свойств, очень легко понять:

```
<Recipe Name="Simple Vegetable Chili">
  <TinOfKidneyBeans Recipe.Quantity="2" Mashed="true" />
  <TinOfChoppedTomatoes Recipe.Quantity="2" />
  <FreshChili Recipe.Quantity="5" Notes="Chopped fine, vary to taste." />
  <Onion Recipe.Quantity="1" Notes="Chopped and fried in olive oil." />
  <LBVPort Notes="Just a dash." />
</Recipe>
```

Синтаксис, использованный здесь — это форма рассмотренного ранее синтаксиса атрибутов. Обращение к присоединенному свойству осуществляется с указанием имени родительского элемента, точки и имени присоединенного свойства.

В WPF присоединенные свойства имеют различные применения. Когда в разделе "Управление компоновкой" пойдет речь о позиционировании элементов управления, будет приведено много примеров присоединенных свойств. Будет показано, как элементы-контейнеры определяют присоединенные свойства, которые позволяют дочерним элементам определять, например, грань контейнера, с которой необходимо стыковаться.

Маршрутизируемые события

Ввиду своей иерархической природы, приложения WPF часто состоят из элементов управления, содержащих другие элементы управления, которые содержат в себе еще какие-то элементы управления, и т.д. *Маршрутизируемое событие* (routed event) — это механизм, посредством которого событие, затрагивающее один элемент управления в иерархии, можно заставить влиять также и на другие события в иерархии, без необходимости написания сложного кода.

Блестящим примером этого может служить ситуация, когда вы позволяете пользователям взаимодействовать с приложениями посредством мыши — весьма распространенная ситуация. Когда пользователь щелкает на кнопке в приложении, обычно необходимо каким-то образом отреагировать на это событие. Один способ сделать это должен быть знаком по разработке с помощью Windows Forms и ASP.NET: нужно предоставить обработчик события кнопки и реагировать в нем на щелчок.

Хотя это может быть и не видно поначалу, такая техника на самом деле достаточно ограничена и может привести к запутанному коду, например, в некоторых приложениях Windows Forms. Причина в том, что необходим некоторый механизм для идентификации элемента управления, который должен реагировать на щелчок кнопкой мыши инициализацией своего события щелчка, и это не может быть сразу очевидно. В простом примере, приведенном здесь, возникает вопрос: должно ли инициироваться событие щелчка на кнопке или же событие щелчка в окне, содержащем кнопку? Если и кнопка, и окно имеют обработчики события, и только одно из этих событий может быть инициировано, скорее всего, это должно быть событие кнопки. Однако что, если нужно, чтобы произошли *оба* события, причем в определенном порядке? В приложении Windows Forms для этого пришлось бы написать (возможно, сложный) специальный код.

Событие щелчка кнопкой мыши для элементов управления WPF, включая `Button` и `Window`, реализовано в виде маршрутизируемого события, что разрешает проблему. Маршрутизируемые события генерируются всеми объектами в иерархии в определенном порядке, предоставляя полный контроль над реакцией на них.

Например, рассмотрим окно `Window`, содержащее элемент `Grid`, который, в свою очередь, содержит `Rectangle`. В результате щелчка на `Rectangle` происходит следующая последовательность событий.

1. В `Window` инициируется событие нажатия кнопки мыши.
2. В `Grid` инициируется событие нажатия кнопки мыши.

3. В `Rectangle` инициируется событие нажатия кнопки мыши (можете показаться, что на этом все заканчивается, но последовательность продолжается).
4. Другое событие нажатия кнопки мыши инициируется в `Rectangle`.
5. Другое событие нажатия кнопки мыши инициируется в `Grid`.
6. Другое событие нажатия кнопки мыши инициируется в `Window`.

Отреагировать на событие можно в любой точке описанной последовательности, добавив соответствующий метод-обработчик. В любой точке метода-обработчика можно также прервать последовательность, но обработчики события не делают этого по умолчанию. Это значит, что можно инициировать несколько методов-обработчиков для одного события (в данном случае — одного события нажатия кнопки мыши).

Для описания приведенной выше последовательности событий в WPF предусмотрена удобная терминология. Когда событие перемещается вниз по иерархии элементов управления, это называется *туннелированием* (tunneling). Прохождение события в противоположном направлении называется *пузырьковым распространением* (bubbling).

Вдобавок всякий раз, когда маршрутизируемые события используются в WPF, имена событий помогают понять, является данное событие туннелируемым или пузырьковым. Все туннелируемые события начинаются с префикса `Preview`. Например, элемент управления `Window` поддерживает события и `PreviewMouseDown`, и `MouseDown`. При желании можно добавить обработчик к одному из этих событий, обоим, либо ни к тому, ни к другому.

На рис. 25.8 показана описанная выше последовательность в терминах событий и моментов их инициации — как происходит туннелирование и пузырьковое распространение по иерархии элементов управления.

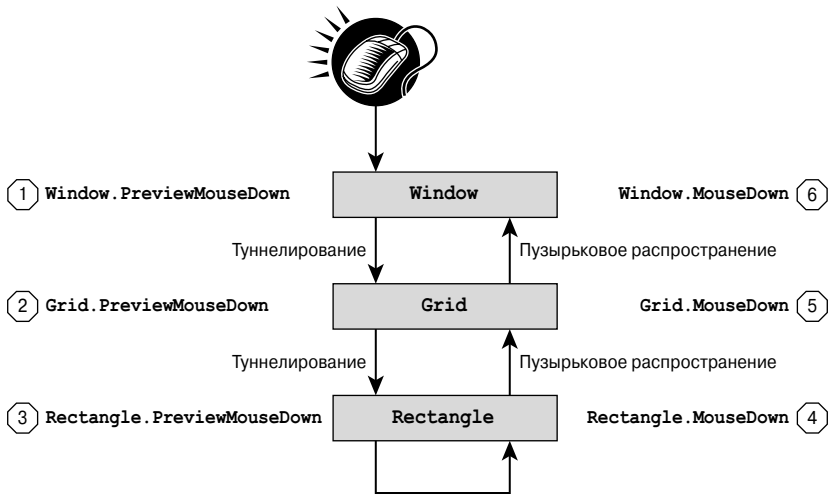


Рис. 25.8. *Туннелирование и пузырьковое распространение событий*

Обработчики маршрутизируемых событий принимают два параметра: источник событие и экземпляр `RoutedEventArgs` или класса-наследника `RoutedEventArgs`.

В реализации обработчика маршрутизируемого события можно установить свойство `Handled` объекта `RoutedEventArgs` в `true`. Если сделать так, то дальнейшая обработка не происходит, т.е. никакие другие обработчики этого события вызываться не будут.

Класс `RoutedEventArgs` также предоставляет свойство по имени `Source`, которое позволяет узнать, какой элемент управления изначально инициировал событие. Это элемент,

в котором среда WPF изначально обнаружила событие, поэтому в сценарии, проиллюстрированном на рис. 25.8, таким элементом будет `Rectangle`. Это очень удобно, поскольку родительский элемент управления может определять, на каком из его дочерних элементов был совершен щелчок. Обратите внимание, что упомянутая "проверка попадания" довольно сложна. WPF может игнорировать щелчки на прозрачных областях элементов управления, не требуя никаких дополнительных действий для этого. В качестве альтернативы можно создавать прозрачные элементы управления, реагирующие на щелчки кнопками мыши, так что платформа предлагает значительную гибкость.

При выполнении проверки попадания WPF различает "прозрачные" и "пустые" области элементов управления. В проверке участвуют только прозрачные области, а пустые области всегда игнорируются.

Маршрутизируемые события охватывают намного больше, чем простые щелчки кнопками мыши; их можно использовать для самых разнообразных целей, включая взаимодействие с клавиатурой, привязку данных, таймеры и многое другое. Присоединенные события, которые мы вскоре рассмотрим, делают маршрутизируемые события еще более полезными.

В следующем практическом занятии иллюстрируется ситуация, описанная в этом разделе, и дается дополнительная информация о маршрутизируемых событиях.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Работа с маршрутизируемыми событиями

1. Создайте приложение WPF по имени `Ch25Ex02` и сохраните его в каталоге `C:\BegVCSharp\Chapter25`.

2. Модифицируйте код в `MainWindow.xaml` следующим образом:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch25Ex02.MainWindow"
  Title="Routed Events" Height="400" Width="800"
  MouseDown="Generic_MouseDown" PreviewMouseDown="Generic_MouseDown"
  MouseUp="Window_MouseUp">
  <Grid Name="contentGrid" MouseDown="Generic_MouseDown"
    PreviewMouseDown="Generic_MouseDown" Background="Azure">
    <Rectangle Name="clickMeRectangle" Margin="10,10,0,0"
      Height="23" HorizontalAlignment="Left" VerticalAlignment="Top"
      Width="70" Stroke="Black" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown" Fill="CadetBlue" />
    <Button Name="clickMeButton" Margin="0,10,10,0" Height="23"
      HorizontalAlignment="Right" VerticalAlignment="Top" Width="70"
      MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown"
      Click="clickMeButton_Click">Click Me</Button>
    <TextBlock Name="outputText" Margin="10,40,10,10"
      Background="Cornsilk" />
  </Grid>
</Window>
```

*Фрагмент кода **Ch25Ex02\MainWindow.xaml***

3. Модифицируйте код в `MainWindow.xaml.cs`, как показано ниже (обратите внимание, что в зависимости от используемой IDE-среды и способа ввода кода XAML, могут быть добавлены автоматически некоторые пустые обработчики):

```
public partial class MainWindow : Window
{
  ...
  private void Generic_MouseDown(object sender,
        MouseButtonEventArgs e)
  {
    outputText.Text = string.Format(
      "{0}Event {1} raised by control {2}. e.Source={3}\n",
      outputText.Text,
      e.RoutedEvent.Name,
      sender.ToString(),
      ((FrameworkElement)e.Source).Name);
  }
  private void Window_MouseUp(object sender, MouseButtonEventArgs e)
  {
    outputText.Text = string.Format(
      "{0}==========\n\n",
      outputText.Text);
  }
  private void clickMeButton_Click(object sender, RoutedEventArgs e)
  {
    outputText.Text = string.Format(
      "{0}Button clicked!\n==========\n\n",
      outputText.Text); // Щелчок на кнопке
  }
}
```

Фрагмент кода ***Ch25Ex02\MainWindow.xaml.cs***

4. Запустите приложение. В работающем приложении щелкните один раз на прямоугольнике в верхнем левом углу, один раз — на светло-голубой области между прямоугольником и кнопкой, и еще один раз — на кнопке. Результат показан на рис. 25.9.

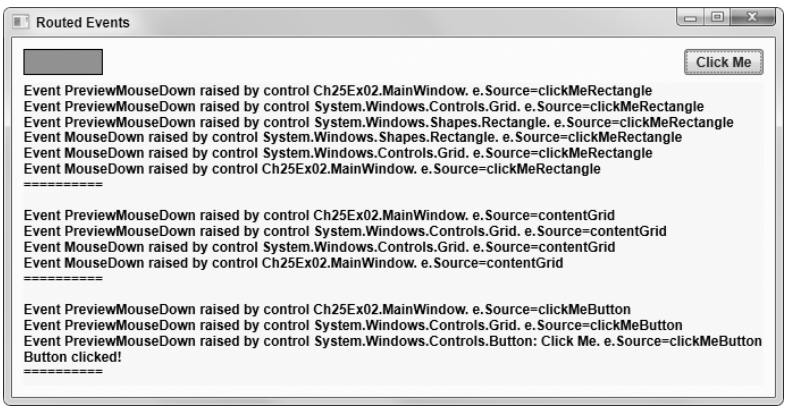


Рис. 25.9. *Обработка маршрутизируемых событий*

Описание работы

В этом примере показано, как обрабатывать маршрутизируемые события, с акцентом на событиях `MouseDown` и `PreviewMouseDown`, поддерживаемых элементами управления WPF. Вы также увидели, что происходит, когда в цепочку событий включена кнопка. Использованный код XAML очень прост, но чтобы выделить важнейшие его части (в контексте данного примера), рассмотрим следующий фрагмент:

```
<Window
  x:Class="Ch25Ex02.MainWindow" MouseDown="Generic_MouseDown"
  PreviewMouseDown="Generic_MouseDown" MouseUp="Window_MouseUp">
  <Grid Name="contentGrid" MouseDown="Generic_MouseDown"
    PreviewMouseDown="Generic_MouseDown">
    <Rectangle Name="clickMeRectangle" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown" />
    <Button Name="clickMeButton" MouseDown="Generic_MouseDown"
      PreviewMouseDown="Generic_MouseDown"
      Click="clickMeButton_Click" />
    <TextBlock Name="outputText" />
  </Grid>
</Window>
```

Здесь все свойства, которые не влияют на функциональность, удалены, поэтому можно сосредоточиться на коде, который относится к обработке маршрутизируемого события. Применяются три обработчика событий, сконфигурированные, как показано в табл. 25.1.

Таблица 25.1. Обработчики и обрабатываемые ими события

Обработчик события	Обрабатываемые события

`Generic_MouseDown()`	`Window.PreviewMouseDown`
	`Window.MouseDown`
	`Grid.PreviewMouseDown`
	`Grid.MouseDown`
	`Rectangle.PreviewMouseDown`
	`Rectangle.MouseDown`

`Window_MouseUp()`	`Window.MouseUp`
`clickMeButton_Click()`	`Button.Click`

Методы-обработчики событий просто выводят информацию в элемент управления `TextBlock`, чтобы можно было видеть, что происходит. Текстовый вывод включает имя события, элемент управления, инициировавший событие, а также исходный элемент управления, полученный из `RoutedEventArgs.Source`.

После запуска приложения первый щелчок на элементе управления `Rectangle` дает последовательность событий, описанную перед практическим занятием. Обработчик событий `Generic_MouseDown()` был вызван шесть раз: три раза для туннелированного события `PreviewMouseDown` и три раза для пузырьковых событий `MouseDown`. Источником события во всех случаях был, как и следовало ожидать, элемент, удовлетворяющий проверке попадания, а именно — `Rectangle` по имени `clickMeRectangle`. Обработчик события `Window_MouseUp()` также был вызван после других обработчиков, добавив некоторый текст для отделения этого теста от следующего.

После этого был произведен щелчок между элементами управления, т.е. фактически — на фоне элемента `Grid`. На этот раз обработчик событий `Generic_MouseDown()` был вызван четыре раза: дважды для туннелированных событий `PreviesMouseDown` и дважды — для пузырьковых событий `MouseDown`. Источником событий на этот раз был элемент `Grid` по имени `contentGrid`. Опять-таки, обработчик `Window_MouseUp()` был вызван после вызовов `Generic_MouseDown()`.

Наконец, был совершен щелчок на элементе управления `Button`. На этот раз `Generic_MouseDown()` был вызван три раза. Затем было инициировано событие `Button.Click`, в результате чего последовал вызов `clickMeButton_Click()`. Наконец, был вызван обработчик `Window_MouseUp()`.

В этой заключительной цепочке событий событие `MouseDown` было обработано кнопкой и использовано для инициализации события `Click`. Применялась реализация

обработчика события MouseDown кнопки для установки свойства Handled аргументов RoutedEventArgs в true. Это прервало поток событий, так что событие MouseDown не вернулось обратно в иерархию элементов управления.

Если возникли вопросы, вернитесь к щелчку на элементе управления Rectangle (перед практическим занятием), чтобы увидеть, как элемент управления Button прерывает маршрутизацию события.

Присоединенные события

В предыдущем примере к элементу управления Button на странице было добавлено событие Button.Click. Для Grid и Window обработчики Click не добавлялись, потому что у этих двух элементов управления просто нет события Click. Однако иногда они могут понадобиться.

Например, предположим, что имеется окно, содержащее 1000 кнопок, и нужно обрабатывать событие Click для каждой из них. Можно предусмотреть 1000 обработчиков событий или же упростить все и создать один общий обработчик событий. Но даже единственный обработчик должен быть ассоциирован с каждым событием Button.Click.

В WPF предлагается альтернативный (и лучший) способ выхода из такой ситуации — *присоединенные события* (attached events). Используя систему присоединенных событий, можно обработать событие Button.Click в Grid, содержащем кнопки, даже несмотря на то, что Grid не имеет события Click. Фактически осуществляется обработка события ButtonBase.Click, т.к. ButtonBase — класс, определяющий событие Click, которое наследует элемент управления Button.

Синтаксис, используемый для этого — тот же синтаксис атрибутов, применяемый для присоединенных свойств:

```
<Grid Name="contentGrid" ButtonBase.Click="contentGrid_Click" ...>
  <Button Name="button1" ...>Button1</Button>
  <Button Name="button2" ...>Button2</Button>
  ...
  <Button Name="button1000" ...>Button1000</Button>
</Grid>
```

Обработчик событий в параметре sender получает ссылку на элемент управления Grid. С помощью свойства RoutedEventArgs.Source можно определить кнопку, на которой был совершен щелчок, чтобы соответствующим образом отреагировать. Это событие инициируется только по щелчку на кнопке, а не когда производится щелчок на фоне элемента управления Grid, потому что элемент Grid не имеет события Click.

Управление компоновкой

До сих пор в этой главе для компоновки элементов управления применялся элемент Grid — в основном потому, что этот элемент предоставляется по умолчанию после создания нового приложения WPF. Однако мы еще не рассматривали все возможности этого класса, как и не говорили о других контейнерах компоновки, которые можно использовать для реализации альтернативных компоновок. В этом разделе управление компоновкой рассматривается более подробно, поскольку это — фундаментальная концепция, которой следует овладеть в WPF.

Все элементы управления компоновкой унаследованы от абстрактного класса Panel. Этот класс просто определяет контейнер, который может содержать коллекцию объектов, унаследованных от UIElement. Все элементы управления WPF наследуются от UIElement. Использовать класс Panel непосредственно для управления компоновкой нельзя, но при желании можно унаследовать от него собственный класс. В качестве альтернативы можно воспользоваться одним из следующих элементов управления, унаследованных от Panel.

- `Canvas`. Этот элемент позволяет позиционировать дочерние элементы так, как вы их видите. Он не накладывает никаких ограничений на расположение дочерних элементов управления, но и не обеспечивает никакой помощи в этом.
- `DockPanel`. Этот элемент управления позволяет пристыковывать дочерние элементы управления к одной из четырех граней. Последний дочерний элемент заполняет оставшееся пространство.
- `Grid`. Вы уже видели, как этот элемент управления позволяет гибко позиционировать дочерние элементы. Чего вы еще не видели — так это как разбить компоновку этого элемента на строки и столбцы, что позволит выравнивать элементы в сетчатой (табличной) компоновке.
- `StackPanel`. Этот элемент управления позиционирует свои дочерние элементы в последовательную вертикальную или горизонтальную компоновку.
- `WrapPanel`. Этот элемент позиционирует свои дочерние элементы в последовательную вертикальную или горизонтальную компоновку, как и `StackPanel`, но вместо единственной строки или столбца, он "переносит" свои дочерние элементы на множество строк или столбцов, в соответствии с доступным пространством.

Скоро будет показано, как использовать эти элементы управления. Но сперва нужно понять несколько базовых концепций.

- Как элементы управления появляются в порядке стека.
- Как использовать выравнивание, поля и дополнения для позиционирования элементов управления и их содержимого.
- Как использовать элемент управления `Border`.

Порядок стека

Когда элемент-контейнер содержит множество дочерних элементов, они визуализируются в определенном порядке стека. Вам должна быть знакома эта концепция по пакетам рисования. Лучший способ воображения порядка стека состоит в том, чтобы представить, что каждый элемент управления находится на стеклянной панели, а контейнер содержит стопку этих панелей. Таким образом, вид контейнера определяется тем, что вы видите, глядя на эту стопку сверху вниз. Элементы, содержащиеся в контейнере, перекрываются, так что то, что вы видите, зависит от порядка расположения этих стеклянных панелей. Если элемент находится в стеке выше, именно его вы увидите в перекрывающейся области. Элементы, находящиеся ниже, могут быть частично или полностью скрыты элементами, расположенными выше.

Это также влияет на проверку попадания, которая осуществляется после щелчка кнопкой мыши в окне. Целевым элементом управления всегда будет тот, что находится выше всех в стеке, когда речь идет о перекрывающихся элементах. Порядок в стеке элементов определяется порядком, в котором они расположены в списке дочерних элементов контейнера. Первый дочерний элемент контейнера размещается на самом нижнем уровне стека, а последний — на самом верхнем. Дочерние элементы, находящиеся между первым и последним, размещаются по порядку уровней — снизу вверх. Как вы вскоре убедитесь, порядок стека элементов управления оказывает дополнительное влияние на некоторые элементы управления компоновкой, используемые в WPF.

Выравнивание, поля, дополнение и размеры

Приведенные ранее примеры продемонстрировали, как с помощью комбинации свойств `Margin`, `HorizontalAlignment` и `VerticalAlignment` позиционировать элементы управления в контейнере `Grid`. Также вы видели, как использовать `Height` и `Width` для указания размеров. Эти свойства, наряду с `Padding`, которое пока не рассматривалось,

полезны во всех элементах управления компоновкой (или в большинстве из них), но поразному. Различные элементы управления компоновкой также могут устанавливать значения по умолчанию для этих свойств. Вы увидите это в примерах последующих разделов, но прежде чем приступить к этому, необходимо разобраться в основах.

Два свойства выравнивания определяют, как элемент выровнен, но мы пока не рассматривали все значения этих свойств. `HorizontalAlignment`, например, можно устанавливать в `Left`, `Right`, `Center` или `Stretch`. Значения `Left` и `Right` позиционируют элементы по левой или правой границе контейнера, `Center` позиционирует элемент посредине, а `Stretch` изменяет ширину элемента управления так, что его боковые границы достигают границ контейнера. Свойство `VerticalAlignment` похоже и принимает значения `Top`, `Bottom`, `Center` или `Stretch`.

С помощью свойств `Margin` и `Padding` задается пространство, которое нужно оставить пустым, соответственно, вдоль границ элемента и внутри границ элемента. В приведенных ранее примерах `Margin` использовалось для позиционирования элементов управления относительно, например, левого верхнего угла `Grid`. Это работало потому, что в случае установки `HorizontalAlignment` в `Left`, а `VerticalAlignment` — в `Top`, элемент управления располагался точно в левом верхнем углу, а установка свойства `Margin` обеспечивала добавление отступа вокруг границ элемента управления. Свойство `Padding` применяется аналогичным образом, но формирует отступы содержимого элемента управления от его границ. Это особенно удобно для `Border`, как будет показано в следующем разделе. Свойства `Padding` и `Margin` могут указываться четырьмя частями (в форме `leftAmount`, `topAmount`, `rightAmount`, `bottomAmount`) или в виде единого значения (`Thickness`).

Позднее вы увидите, что `Height` и `Width` часто управляются другими свойствами. Например, при значении `HorizontalAlignment` равном в `Stretch`, свойство `Width` элемента управления изменяется при изменении ширины его контейнера.

Элемент управления `Border`

`Border` — очень простой и полезный контейнерный элемент управления. Он содержит единственный дочерний элемент, а не множество, как более сложные элементы, которые рассматриваются ниже. Размеры дочернего элемента определяются так, чтобы полностью заполнить элемент `Border`. Это может показаться не особенно удобным, но не забывайте, что вы всегда можете использовать свойства `Margin` и `Padding` для позиционирования `Border` внутри его контейнера, а содержимого `Border` — внутри границ `Border`. Можно, например, установить свойство `Background` элемента `Border` так, чтобы он был видимым. Чуть позже вы увидите этот элемент управления в действии.

Элемент управления `Canvas`

Элемент управления `Canvas`, как упоминалось ранее, предоставляет полную свободу в расположении элементов управления. Другая особенность `Canvas` состоит в том, что свойства `HorizontalAligment` и `VerticalAlignment`, применяемые с дочерними элементами, не оказывают влияния на расположение этих элементов.

Свойство `Margin` можно использовать для позиционирования элементов в `Canvas`, как и в предыдущих примерах, но лучше применять присоединенные свойства `Canvas.Left`, `Canvas.Top`, `Canvas.Right` и `Canvas.Bottom`, которые предоставляются классом `Canvas`:

```
<Canvas ...>
  <Button Canvas.Top="10" Canvas.Left="10" ...>Button1</Button>
</Canvas>
```

Этот код позиционирует элемент `Button` так, что его верхняя грань отстоит на 10 пикселей от границы `Canvas`, а левая грань — на 10 пикселей от левой грани `Canvas`. Обратите внимание, что свойства `Top` и `Left` имеют приоритет перед `Bottom` и `Right`. Например, если вы укажете и `Top`, и `Bottom`, то свойство `Bottom` игнорируется.

На рис. 25.10 показаны два элемента управления Rectangle, позиционированные в элементе Canvas, с окном, размер которого изменен по двум измерениям.

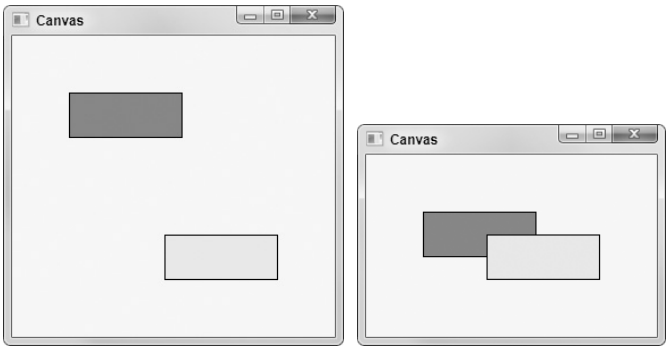


***Рис. 25.10**. Использование элемента Canvas*

Все примеры компоновок из этого раздела можно найти в проекте LayoutExamples внутри каталога кода для этой главы.

Один элемент Rectangle позиционирован относительно верхнего левого угла, а другой — относительно нижнего правого угла. При изменении размера окна эти относительные положения сохраняются. Здесь также можно убедиться в важности порядка стека элементов управления Rectangle. Нижний правый Rectangle находится выше в порядке стека, поэтому именно его вы видите, когда два эти элемента перекрываются.

Ниже приведен код этого примера:

```
<Canvas Background="AliceBlue">
  <Rectangle Canvas.Left="50" Canvas.Top="50" Height="40" Width="100"
    Stroke="Black" Fill="Chocolate" />
  <Rectangle Canvas.Right="50" Canvas.Bottom="50" Height="40" Width="100"
    Stroke="Black" Fill="Bisque" />
</Canvas>
```

*Фрагмент кода **LayoutExamples\CanvasWindow.xaml***

Элемент управления DockPanel

Элемент управления DockPanel позволяет пристыковывать элементы управления к одной из своих граней. Такого рода компоновка должна быть знакомой, даже если вы никогда не слышали о ней ранее. Именно так, например, элемент управления Ribbon остается наверху окна Microsoft Word, и именно так позиционируются различные окна в средах VS и VCE. В VS и VCE можно также (намеренно или случайно) изменить место стыковки окна, перетащив его с места на место.

DockPanel имеет единственное присоединенное свойство, которое могут использовать дочерние элементы для указания границы контейнера, к которой элементы пристыковываются: DockPanel.Dock. Это свойство может быть установлено в Left, Top, Right или Bottom.

Порядок стека элементов управления в DockPanel чрезвычайно важен, поскольку всякий раз, когда элемент управления пристыковывается к границе контейнера, тем самым сокращается доступное пространство для последующих дочерних элементов управления. Например, панель инструментов можно пристыковать к верхней грани DockPanel, а затем вторую панель инструментов — к левой стороне DockPanel. Первый элемент растянется на всю ширину верхней области отображения DockPanel, а второй займет место только от нижней границы до первой панели инструментов вдоль левой границы DockPanel.

Последний из указываемых дочерних элементов (обычно) заполняет область, оставшуюся после позиционирования предыдущего дочернего элемента (этим поведением можно управлять).

При позиционировании элемента управления в `DockPanel` область, занятая элементом управления, может быть меньше, чем область в `DockPanel`, которая зарезервирована для элемента. Например, если вы пристыкуете `Button` со значением `Width` 100 и `Height` 50, значением `HorizontalAlignment`, равным `Left`, в верхнюю часть `DockPanel`, то справа от `Button` останется пространство, не использованное другими пристыкованными дочерними элементами. Вдобавок, если элемент `Button` имеет значение свойства `Margin`, равное `20`, то всего будет зарезервировано 90 пикселей в верхней части `DockPanel` (высота элемента управления плюс верхнее и нижнее поля). Это поведение следует принимать во внимание, когда используется `DockPanel` для компоновки; в противном случае результаты могут быть неожиданными.

На рис. 25.11 показан пример компоновки `DockPanel`

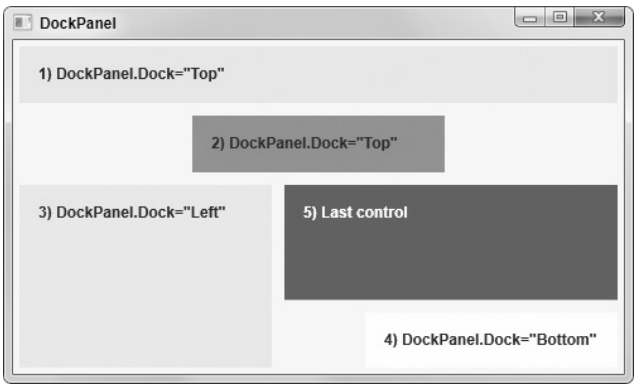


***Рис. 25.11**. Пример компоновки `DockPanel`*

Код для этой компоновки выглядит следующим образом:

```
<DockPanel Background="AliceBlue">
  <Border DockPanel.Dock="Top" Padding="10" Margin="5"
    Background="Aquamarine" Height="45">
    <Label>1) DockPanel.Dock="Top"</Label>
  </Border>
  <Border DockPanel.Dock="Top" Padding="10" Margin="5"
    Background="PaleVioletRed" Height="45" Width="200">
    <Label>2) DockPanel.Dock="Top"</Label>
  </Border>
  <Border DockPanel.Dock="Left" Padding="10" Margin="5"
    Background="Bisque" Width="200">
    <Label>3) DockPanel.Dock="Left"</Label>
  </Border>
  <Border DockPanel.Dock="Bottom" Padding="10" Margin="5"
    Background="Ivory" Width="200" HorizontalAlignment="Right">
    <Label>4) DockPanel.Dock="Bottom"</Label>
  </Border>
  <Border Padding="10" Margin="5" Background="BlueViolet">
    <Label Foreground="White">5) Last control</Label>
  </Border>
</DockPanel>
```

*Фрагмент кода **LayoutExamples\DockPanelWindow.xaml***

В этом коде используется элемент управления Border для четкой пометки областей стыковки элементов управления в примере компоновки, а также элементы управления Label для вывода простого информативного текста. Чтобы понять компоновку, вы должны читать ее сверху вниз, просматривая все элементы управления по очереди.

1. Первый элемент управления Border пристыкован к верхней грани DockPanel. Общая область, занятая у DockPanel, составила 55 пикселей (Height+2×Margin). Обратите внимание, что свойство Padding не влияет на компоновку, поскольку оно действует внутри Border, однако это свойство управляет позицией встроенного элемента управления Label. Элемент управления Border заполняет все доступное место до границы, к которой он пристыкован, если не ограничен свойствами Height и Width, и потому он растянут на всю DockPanel.
2. Второй элемент управления Border также пристыкован к верхней грани DockPanel и занимает еще 55 пикселей от верха отображаемой области. Этот элемент управления Border также включает свойство Width, которое заставляет Border занимать только часть ширины DockPanel. Он позиционируется по центру, поскольку значение по умолчанию для HorizontalAlignment в DockPanel равно Center.
3. Третий элемент управления Border пристыкован к левой границе DockPanel и занимает 210 пикселей в левой части отображаемой области.
4. Четвертый элемент управления Border пристыкован к низу DockPanel и занимает 20 пикселей плюс высота элемента Label, который он содержит. Высота определяется свойствами Margin, Padding и содержимым элемента управления Border, т.к. не указана явно. Элемент Border пристыкован к нижнему правому углу DockPanel, поскольку его HorizontalAlignment равно Right.
5. Пятый, и последний, элемент управления Border заполняет оставшееся пространство.

Запустите этот пример и поэкспериментируйте с изменением размера содержимого. Обратите внимание, что чем выше положение элемента управления в стеке, тем больший приоритет отдается его пространству. При сокращении окна пятый элемент Border может быстро быть перекрыт элементами с более высоким порядком в стеке. Во избежание этого, будьте осторожны при использовании элемента управления компоновкой DockPanel, возможно, устанавливая минимальный размер для окна.

Элемент управления Grid

Элемент управления Grid может иметь множество строк и столбцов, которые служат для размещения дочерних элементов. В этой главе Grid уже использовался несколько раз, но во всех случаях — с единственной строкой и столбцом. Для добавления дополнительных строк и столбцов применяются свойства RowDefinitions и ColumnDefinitions, представляющие собой коллекции объектов RowDefinition и ColumnDefinition. Эти объекты должны указываться в синтаксисе элемента свойства, например:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  ...
</Grid>
```

В этом коде определен элемент Grid из трех строк и двух столбцов. Обратите внимание, что здесь не требуется никакой дополнительной информации: каждая строка и столбец динамически изменяет размеры при изменении размеров Grid. Каждая строка занимает треть от высоты Grid, а каждый столбец — половину его ширины. Можно отобразить линии между ячейками в Grid, устанавливая свойство Grid.ShowGridlines в true.

Изменением размеров можно управлять с помощью свойств Width, Height, MinWidth, MaxWidth, MinHeight и MaxHeight. Например, установка свойства Width столбца гарантирует, что он сохранит эту ширину. Установка для свойства Width значения * означает "заполнить оставшееся пространство после вычисления ширин всех прочих столбцов". Это установка по умолчанию. При наличии нескольких столбцов с Width, равным *, оставшееся место делится между ними поровну. Значение * также может применяться для установки свойства строк Height. Другим возможным значением для Height и Width является Auto, которое устанавливает размеры строки и столбца в соответствии с их содержимым. Также можно применять элементы управления GridSplitter для разрешения пользователям устанавливать размеры строк и столбцов с помощью щелчков и перетаскивания.

Дочерние элементы управления элемента Grid могут использовать присоединенные свойства Grid.Column и Grid.Row, чтобы указывать ячейку, в которой они размещаются. По умолчанию оба эти свойства равны 0, так что если вы пропустите их, то дочерний элемент будет помещен в самую верхнюю левую ячейку. Дочерние элементы управления также могут использовать Grid.ColumnSpan и Grid.RowSpan для позиционирования в нескольких ячейках таблицы, где верхняя левая ячейка указывается свойствами Grid.Column и Grid.Row.

На рис. 25.12 показан элемент управления Grid, содержащий несколько эллипсов и элемент GridSplitter с окном, увеличенным вдвое.

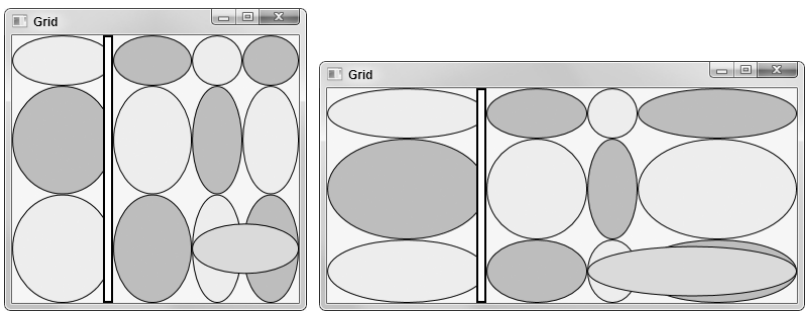


***Рис. 25.12**. Пример элемента управления Grid*

Ниже приведен использованный для этого код:

```
<Grid Background="AliceBlue">
  <Grid.ColumnDefinitions>
    <ColumnDefinition MinWidth="100" MaxWidth="200" />
    <ColumnDefinition MaxWidth="100" />
    <ColumnDefinition Width="50" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition MinHeight="100" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Ellipse Grid.Row="0" Grid.Column="0" Fill="BlanchedAlmond"
    Stroke="Black" />
```

```
    <Ellipse Grid.Row="0" Grid.Column="1" Fill="BurlyWood"
      Stroke="Black" />
    <Ellipse Grid.Row="0" Grid.Column="2" Fill="BlanchedAlmond"
      Stroke="Black" />
    <Ellipse Grid.Row="0" Grid.Column="3" Fill="BurlyWood"
      Stroke="Black" />
    <Ellipse Grid.Row="1" Grid.Column="0" Fill="BurlyWood"
      Stroke="Black" />
    <Ellipse Grid.Row="1" Grid.Column="1" Fill="BlanchedAlmond"
      Stroke="Black" />
    <Ellipse Grid.Row="1" Grid.Column="2" Fill="BurlyWood"
      Stroke="Black" />
    <Ellipse Grid.Row="1" Grid.Column="3" Fill="BlanchedAlmond"
      Stroke="Black" />
    <Ellipse Grid.Row="2" Grid.Column="0" Fill="BlanchedAlmond"
      Stroke="Black" />
    <Ellipse Grid.Row="2" Grid.Column="1" Fill="BurlyWood"
      Stroke="Black" />
    <Ellipse Grid.Row="2" Grid.Column="2" Fill="BlanchedAlmond"
      Stroke="Black" />
    <Ellipse Grid.Row="2" Grid.Column="3" Fill="BurlyWood"
      Stroke="Black" />
    <Ellipse Grid.Row="2" Grid.Column="2" Grid.ColumnSpan="2" Fill="Gold"
      Stroke="Black" Height="50"/>
    <GridSplitter Grid.RowSpan="3" Width="10" BorderThickness="2">
      <GridSplitter.BorderBrush>
        <SolidColorBrush Color="Black" />
      </GridSplitter.BorderBrush>
    </GridSplitter>
  </Grid>
```

Фрагмент кода ***LayoutExamples\GridWindow.xaml***

Этот код использует различные комбинации свойств в определениях строки и столбца для достижения интересного эффекта при изменении размера отображения, так что вам стоит опробовать их самостоятельно.

Сначала рассмотрим строки. Верхняя строка имеет фиксированную высоту в 50 пикселей, вторая строка — минимальную высоту в 100 пикселей, а третья строка заполняет оставшееся пространство. Это значит, что если `Grid` имеет высоту менее 150 пикселей, то третья строка не будет видимой. Когда `Grid` имеет высоту от 150 до 250 пикселей, изменяется только размер третьей строки — от 0 до 100 пикселей. Причина в том, что оставшееся пространство вычисляется как общая высота минус общая высота строк, имеющих фиксированную высоту. Оставшееся пространство распределяется между второй и третьей строками, но поскольку вторая строка имеет минимальную высоту в 100 пикселей, она не изменит свою высоту до тех пор, пока общая высота `Grid` не достигнет 250 пикселей. Наконец, когда высота `Grid` больше 250, вторая и третья строки разделят оставшееся пространство, чтобы их общая высота была равна или больше 100 пикселей.

Теперь рассмотрим столбцы. Только третий столбец имеет фиксированный размер в 50 пикселей. Первый и второй столбцы занимают вместе до 300 пикселей. Поэтому только четвертый столбец будет увеличиваться в размерах, когда общая ширина `Grid` превысит 550 пикселей. Чтобы понять это, представьте, сколько пикселей доступно столбцам и как они распределяются. Сначала 50 пикселей отводятся третьему столбцу, оставляя 500 остальным столбцам. Третий столбец имеет максимальную ширину в 100 пикселей, что оставляет 400 для первого и четвертого. Первый столбец имеет максимум 200 пикселей, так что если ширина превысит этот предел, он не займет больше места. Вместо этого четвертый столбец увеличится в размере.

Обратите внимание на два дополнительных момента в этом примере. Во-первых, последний эллипс охватывает третий и четвертый столбец для иллюстрации свойства `Grid.ColumnSpan`. Во-вторых, предоставлен элемент `GridSplitter`, который позволяет изменять размер первого и второго столбцов. Однако, как только общая ширина элемента `Grid` превысит 550 пикселей, `GridSplitter` не сможет изменять размеры этих столбцов, поскольку ни первый, ни второй столбцы не смогут увеличиваться в размерах.

Элемент управления `GridSpitter` полезен, но имеет очень непритязательный вид. Это один из элементов, которым действительно нужна стилизация, или, по крайней мере, невидимость, за счет установки свойства `Background` в `Transparent`.

При наличии в окне нескольких элементов `Grid` можно также определить общие размерные группы для строк или столбцов, используя свойство `SharedSizeGroup` в определениях строк или столбцов, в котором просто устанавливается произвольный строковый идентификатор. Например, если размер столбца в разделяемой размерной группе изменяется в одном элементе `Grid`, то и столбец из другого `Grid`, но, принадлежащий той же размерной группе, изменится для соответствия этому размеру. Данная функциональность включается и отключается через свойство `Grid.IsSharedSizeScope`.

Элемент управления `StackPanel`

После сложности `Grid` элемент `StackPanel` покажется достаточно простым элементом управления компоновкой. Его можно воспринимать как усеченную версию `DockPanel`, где грань, к которой пристыковываются дочерние элементы, фиксирована для этих элементов. Другое отличие между этими элементами состоит в том, что последний дочерний элемент `StackPanel` не заполняет оставшееся пространство. Однако элементы управления по умолчанию растягиваются до границ элемента управления `StackPanel`.

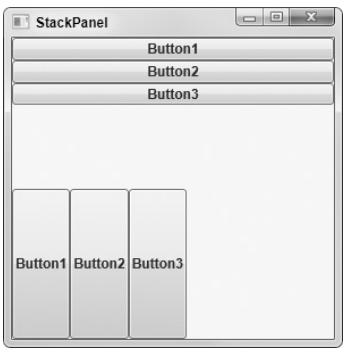


***Рис. 25.13**. Пример элементов управления `StackPanel`*

Направление, в котором располагаются элементы управления, определяется тремя свойствами. Свойство `Orientation` может быть установлено в `Horizontal` или `Vertical`, а `HorizontalAlignment` и `VerticalAlignment` служат для определения того, возле какой границы `StackPanel` располагается каждый элемент: верхней, нижней, левой или правой. Можно даже установить стек элементов в центре `StackPanel`, используя значение `Center` для свойства выравнивания.

На рис. 25.13 показаны два элемента управления `StackPanel`, каждый из которых содержит три кнопки. Элементы управления `StackPanel` позиционируются с использованием элемента `Grid` с двумя строками и одним столбцом.

Используемый здесь код выглядит следующим образом:

```
<Grid Background="AliceBlue">
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <StackPanel Grid.Row="0">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
  <StackPanel Grid.Row="1" Orientation="Horizontal">
    <Button>Button1</Button>
```

```
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
</Grid>
```

Фрагмент кода ***LayoutExamples\StackPanelWindow.xaml***

При использовании компоновки `StackPanel` часто придется добавлять линейки прокрутки, чтобы можно было видеть все элементы управления, содержащиеся в `StackPanel`. Это еще одна область, где WPF берет на себя всю рутинную работу. Для достижения этого можно воспользоваться элементом управления `ScrollViewer`; просто включите `StackPanel` в этот элемент управления:

```
<Grid Background="AliceBlue">
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>

  <ScrollViewer>
    <StackPanel Grid.Row="0">
      <Button>Button1</Button>
      <Button>Button2</Button>
      <Button>Button3</Button>
    </StackPanel>
  </ScrollViewer>

  <StackPanel Grid.Row="1" Orientation="Horizontal">
    <Button>Button1</Button>
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
</Grid>
```

Доступны более сложные приемы для прокрутки, а также программная прокрутка, но часто элемента управления `ScrollViewer` оказывается вполне достаточно.

Элемент управления `WrapPanel`

Элемент управления `WrapPanel` — это, по сути, расширенная версия `StackPanel`; элементы управления, которые не умещаются, переносятся в дополнительные строки (или столбцы). На рис. 25.14 показан элемент управления `WrapPanel`, содержащий несколько фигур в окне изменяющегося размера.

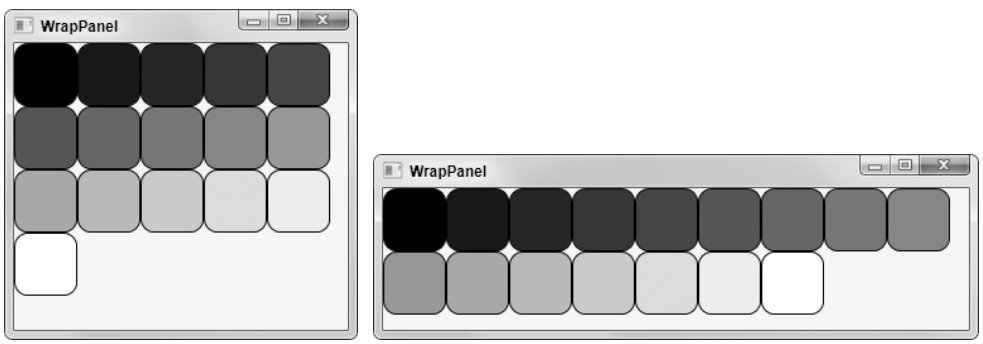


Рис. 25.14*. Пример использования элемента управления* `WrapPanel`

Сокращенная версия необходимого кода показана ниже:

```
<WrapPanel Background="AliceBlue">
  <Rectangle Fill="#FF000000" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  <Rectangle Fill="#FF111111" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  <Rectangle Fill="#FF222222" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
  ...
  <Rectangle Fill="#FFFFFFFF" Height="50" Width="50" Stroke="Black"
    RadiusX="10" RadiusY="10" />
</WrapPanel>
```

Фрагмент кода ***LayoutExamples\WrapPanelWindow.xaml***

Элементы управления `WrapPanel` предоставляют отличный способ создания динамической компоновки, которая позволяет пользователям тонко управлять отображением содержимого.

Стилизация элементов управления

Одним из замечательных свойств WPF является полный контроль, который предоставляется дизайнерам в отношении внешнего вида и поведения пользовательского интерфейса. В центре этого лежит возможность произвольной стилизации элементов управления — в двух или трех измерениях. До сих пор применялась базовая стилизация элементов, появившаяся в .NET 3.5, но существующие на текущий момент возможности просто безграничны.

В этом разделе рассматриваются две базовых технологии.

- **Стили.** Набор свойств, применяемых к элементу управления целиком.
- **Шаблоны.** Элементы управления, используемые для построения отображения элемента.

Между ними существует некоторое перекрытие, поскольку стили могут содержать шаблоны.

Стили

Элементы управления WPF имеют свойство по имени `Style` (унаследованное от `FrameworkElement`), которое может быть установлено в экземпляр класса `Style`. Класс `Style` довольно сложен и включает расширенную функциональность стилизации, но в центре его лежит набор объектов `Setter`. Каждый `Setter` отвечает за установку свойства в соответствии со значением его свойства `Property` (имя свойства для установки) и его свойства `Value` (значение устанавливаемого свойства). Можно либо полностью квалифицировать имя, используемое в `Property` для управления типом (например, `Button.Foreground`), либо устанавливать свойство `TargetType` объекта `Style` (например, `Button`), чтобы он мог определять имена свойств.

В следующем коде показано, как использовать объект `Style` для установки свойства `Foreground` элемента управления `Button`:

```
<Button>
  Click me!
  <Button.Style>
    <Style TargetType="Button">
      <Setter Property="Foreground">
        <Setter.Value>
          <SolidColorBrush Color="Purple" />
        </Setter.Value>
      </Setter>
```

```
    </Style>
  </Button.Style>
</Button>
```

Очевидно, что в этом случае было бы намного легче просто установить свойство `Foreground` кнопки обычным образом. Стили становятся намного полезнее после их превращения в ресурсы, поскольку ресурсы можно использовать повторно. Далее в этой главе будет рассказано, как это делается.

Шаблоны

Элементы управления конструируются с использованием шаблонов, которые можно настраивать. Шаблон состоит из иерархии элементов управления, применяемых для построения отображения элемента, которое может включать презентатор содержимого для таких элементов, как кнопки, отображающие содержимое.

Шаблон элемента управления сохраняется в свойстве `Template`, которое представляет собой экземпляр класса `ControlTemplate`. Класс `ControlTemplate` имеет свойство `TargetType`, позволяющее установить тип элемента управления, для которого определяется шаблон, и он может содержать единственный элемент управления. Этот элемент может быть контейнером, подобным `Grid`, так что на самом деле нет ограничений того, что можно делать.

Обычно шаблон для класса устанавливается с помощью стиля. Это просто включает предоставление элементов управления для использования в свойстве `Template` следующим образом:

```
<Button>
  Click me!
  <Button.Style>
    <Style TargetType="Button">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="Button">
            ...
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </Button.Style>
</Button>
```

Некоторые элементы управления требуют более чем одного шаблона. Например, элементы `CheckBox` используют один шаблон для флажка (`CheckBox.Template`) и один — для текста, выводимого возле флажка (`CheckBox.ContentTemplate`).

Шаблоны, которые требуют презентатора содержимого, могут включать элемент управления `ContentPresenter` в месте, куда будет направлен вывод содержимого. Некоторые элементы управления, в частности те, что выводят коллекции элементов, используют альтернативные приемы, которые в этой главе не рассматриваются.

Опять-таки, замена шаблонов наиболее удобна, когда она комбинируется с ресурсами. Но поскольку стилизация элементов управления — очень распространенная техника, ее стоит опробовать в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Стилизация и шаблоны

1. Создайте приложение WPF по имени `Ch25Ex03` и сохраните его в каталоге `C:\BegVCSharp\Chapter25`.

2. Модифицируйте код в `MainWindow.xaml`, как показано ниже:

```
<Window x:Class="Ch25Ex03.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Nasty Button" Height="150" Width="550">
  <Grid Background="Black">
    <Button Margin="20" Click="Button_Click">
      Would anyone use a button like this?
      <Button.Style>
        <Style TargetType="Button">
          <Setter Property="FontSize" Value="18" />
          <Setter Property="FontFamily" Value="arial" />
          <Setter Property="FontWeight" Value="bold" />
          <Setter Property="Foreground">
            <Setter.Value>
              <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                <LinearGradientBrush.GradientStops>
                  <GradientStop Offset="0.0" Color="Purple" />
                  <GradientStop Offset="0.5" Color="Azure" />
                  <GradientStop Offset="1.0" Color="Purple" />
                </LinearGradientBrush.GradientStops>
              </LinearGradientBrush>
            </Setter.Value>
          </Setter>
          <Setter Property="Template">
            <Setter.Value>
              <ControlTemplate TargetType="Button">
                <Grid>
                  <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="50" />
                  <ColumnDefinition />
                  <ColumnDefinition Width="50" />
                  </Grid.ColumnDefinitions>
                  <Grid.RowDefinitions>
                    <RowDefinition MinHeight="50" />
                  </Grid.RowDefinitions>
                  <Ellipse Grid.Column="0" Height="50">
                    <Ellipse.Fill>
                      <RadialGradientBrush>
                        <RadialGradientBrush.GradientStops>
                          <GradientStop Offset="0.0" Color="Yellow" />
                          <GradientStop Offset="1.0" Color="Red" />
                        </RadialGradientBrush.GradientStops>
                      </RadialGradientBrush>
                    </Ellipse.Fill>
                  </Ellipse>
                  <Grid Grid.Column="1">
                    <Rectangle RadiusX="10" RadiusY="10">
                      <Rectangle.Fill>
                        <RadialGradientBrush>
                          <RadialGradientBrush.GradientStops>
                            <GradientStop Offset="0.0" Color="Yellow" />
                            <GradientStop Offset="1.0" Color="Red" />
                          </RadialGradientBrush.GradientStops>
                        </RadialGradientBrush>
                      </Rectangle.Fill>
                    </Rectangle>
                    <ContentPresenter Margin="20,0,20,0"
                      HorizontalAlignment="Center"
                      VerticalAlignment="Center" />
                  </Grid>
```

```
                <Ellipse Grid.Column="2" Height="50">
                  <Ellipse.Fill>
                    <RadialGradientBrush>
                    <RadialGradientBrush.GradientStops>
                    <GradientStop Offset="0.0" Color="Yellow" />
                    <GradientStop Offset="1.0" Color="Red" />
                    </RadialGradientBrush.GradientStops>
                    </RadialGradientBrush>
                  </Ellipse.Fill>
                </Ellipse>
              </Grid>
            </ControlTemplate>
          </Setter.Value>
        </Setter>
      </Style>
    </Button.Style>
  </Button>
 </Grid>
</Window>
```

Фрагмент кода ***Ch25Ex03\MainWindow.xaml***

3. Модифицируйте код в `MainWindow.xaml.cs` следующим образом:

```
public partial class MainWindow : Window
{
  ...
  private void Button_Click(object sender, RoutedEventArgs e)
  {
    MessageBox.Show("Button clicked.");
  }
}
```

Фрагмент кода ***Ch25Ex03\MainWindow.xaml.cs***

4. Запустите приложение и щелкните на кнопке. Результат показан на рис. 25.15.

Рис. 25.15*. Приложение* `Ch25Ex03` *в работе*

Описание работы

Кнопка в этом примере выглядит не особо привлекательно. Однако оставим пока эстетические соображения и обратим внимание на то, что пример демонстрирует возможность полного изменения внешнего вида кнопки без особых усилий. При смене шаблона функциональность кнопки остается неизменной. То есть на кнопке можно щелкать и в обработчике события реагировать на щелчок.

Возможно, вы заметили, что определенные вещи, которые ассоциируются с кнопками Windows, в используемом здесь шаблоне не реализованы. В частности, нет визуального от-

клика при наведении курсора на кнопку или при щелке на ней. Кроме того, кнопка выглядит одинаково, независимо от того, имеет она фокус или нет. Чтобы добавить эти недостающие эффекты, следует познакомиться с *триггерами*, которые будут темой следующего раздела.

Перед тем, как сделать это, давайте рассмотрим код более подробно, сосредоточив внимание на стилях и шаблонах, а также на том, как шаблон был создан.

Код начинается с обычного фрагмента, который нужен для отображения элемента управления `Button`:

```
<Button Margin="20" Click="Button_Click">
  Would anyone use a button like this?
```

Это устанавливает базовые свойства и содержимое кнопки. Затем в свойстве `Style` устанавливается объект `Style`, который начинается с настройки трех простых свойств шрифта для элемента управления `Button`:

```
<Button.Style>
  <Style TargetType="Button">
    <Setter Property="FontSize" Value="18" />
    <Setter Property="FontFamily" Value="arial" />
    <Setter Property="FontWeight" Value="bold" />
```

Далее устанавливается свойство `Button.Foreground` с использованием синтаксиса элементов, поскольку здесь применяется кисть:

```
<Setter Property="Foreground">
  <Setter.Value>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
      <LinearGradientBrush.GradientStops>
        <GradientStop Offset="0.0" Color="Purple" />
        <GradientStop Offset="0.5" Color="Azure" />
        <GradientStop Offset="1.0" Color="Purple" />
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Setter.Value>
</Setter>
```

Остаток кода для объекта `Style` устанавливает свойство `Button.Template` в объект `ControlTemplate`:

```
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="Button">
            ...
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </Button.Style>
</Button>
```

Код шаблона может быть подытожен как элемент управления `Grid`, содержащий три ячейки в одной строке. Эти ячейки по очереди содержат `Ellipse`, `Rectangle`, наряду с `ContentPresenter` для шаблона, и еще один `Ellipse`:

```
<Grid>
  <Ellipse Grid.Column="0" Height="50">
    ...
  </Ellipse>
  <Grid Grid.Column="1">
    <Rectangle RadiusX="10" RadiusY="10">
      ...
    </Rectangle>
```

```
    <ContentPresenter Margin="20,0,20,0"
      HorizontalAlignment="Center"
      VerticalAlignment="Center" />
  </Grid>
  <Ellipse Grid.Column="2" Height="50">
    ...
  </Ellipse>
</Grid>
```

В коде нет ничего особенно сложного, так что можете проанализировать его дальше самостоятельно.

Триггеры

В первом примере этой главы было показано, как триггеры используются для привязки событий к действиям. События в WPF могут включать самые разные вещи, в том числе щелчки на кнопках, события запуска и останова приложения и т.д. В WPF имеется несколько типов триггеров, и все они унаследованы от класса `TriggerBase`. Тип триггера, показанного в данном примере — `EventTrigger`. Класс `EventTrigger` содержит коллекцию действий, каждое из которых — это объект, унаследованный от базового класса `TriggerAction`. Эти действия выполняются при активизации триггера.

В WPF очень немного классов наследуется от `TriggerAction`. Но, конечно же, есть возможность определять собственные классы. Можно применять `EventTrigger` для запуска анимации с использованием действия `BeginStoryboard`, манипулировать раскадровками с помощью `ControllableStoryboardAction` и инициировать звуковые эффекты посредством `SoundPlayerAction`. Поскольку этот последний триггер наиболее полезен при анимации, мы вернемся к нему в следующем разделе.

Каждый элемент управления имеет свойство `Triggers`, которое служит для определения триггеров непосредственно в элементе управления. Триггеры можно также определять и выше по иерархии, например, в объекте `Window`, как было показано ранее. Тип триггера, чаще всего используемого при стилизации элементов управления, является `Trigger` (хотя для запуска анимации все равно будет применяться `EventTrigger`). Класс `Trigger` используется для установки свойств в ответ на изменения в других свойствах, и он особенно полезен, когда применяется вместе с объектами `Style`.

Объекты триггеров конфигурируются следующим образом.

- Для определения свойства, которое должен отслеживать `Trigger`, используется свойство `Trigger.Property`.
- Для определения времени активизации объекта `Trigger` устанавливается свойство `Trigger.Value`.
- Для определения действий, выполняемых `Trigger`, свойство `Trigger.Setters` устанавливается в коллекцию объектов `Setter`.

Упомянутые здесь объекты `Setter` — это те самые объекты, что были описаны ранее в разделе “Стили”.

Например, следующий триггер проверяет значение свойства по имени `MyBooleanValue`, и когда оно принимает значение `true`, устанавливает значение свойства `Opacity` в `0.5`:

```
<Trigger Property="MyBooleanValue" Value="true">
  <Setter Property="Opacity" Value="0.5" />
</Trigger>
```

Сам по себе этот код не дает особо много, поскольку он не ассоциирован ни с каким элементом управления или стилем. Приведенный ниже код намного более выразителен, поскольку показывает `Trigger`, который используется в объекте `Style`:

```
<Style TargetType="Button">
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="true">
      <Setter Property="Foreground" Value="Yellow" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Этот код должен изменить значение свойства `Foreground` элемента управления `Button` на `Yellow`, когда свойство `Button.IsMouseOver` устанавливается в `true`. `IsMouseOver` — одно из исключительно полезных свойств, которое можно применять в качестве сокращения, чтобы находить информацию об элементах управления и их состоянии. Как следует из имени, оно равно `true`, когда курсор мыши находится над элементом. Это позволяет кодировать динамические изменения, связанные с курсором мыши. К другим свойствам подобного рода относятся: `IsFocused`, позволяющее определить, имеется ли фокус у элемента управления; `IsHitTestVisible`, указывающее, можно ли щелкать на элементе управления (т.е. не скрыт ли данный элемент под другими в порядке стека); `IsPressed`, указывающее на то, нажата ли кнопка. Последнее применимо только к кнопкам, унаследованным от `ButtonBase`, в то время как другие свойства доступны во всех элементах управления.

Как и со свойством `Style.Triggers`, многого можно также достичь, используя свойство `ControlTemplate.Triggers`. Оно позволяет создавать шаблоны, способные реагировать на наведение курсора мыши, щелчки и изменения фокуса. Это также то, что должно модифицироваться при самостоятельной реализации функциональности.

Анимация

Анимации создаются с применением раскадровок. Самый лучший способ создания анимаций, без сомнения, предусматривает использование такого инструмента, как Expression Blend. Тем не менее, определять анимации можно за счет непосредственного редактирования кода XAML, также из отделенного кода (поскольку XAML — это просто способ построения объектной модели WPF).

Раскадровка определяется через объект `Storyboard`, содержащий одну или более *временных шкал* (timeline). Временные шкалы можно определять с использованием ключевых кадров либо несколько более простых объектов, инкапсулирующих всю анимацию. Сложные раскадровки могут даже содержать в себе вложенные раскадровки.

Как показано в примере, `Storyboard` содержится в словаре ресурсов, так что он должен быть идентифицирован свойством `x:Key`.

Внутри временной шкалы раскадровки можно осуществлять анимацию свойств любого элемента в приложении, относящихся к типам `double`, `Point` или `Color`. Это покрывает большинство вещей, которые может понадобиться изменять, так что обеспечивается достаточная гибкость. Есть вещи, которые сделать не удастся, например, полностью заменить одну кисть другой, но с использованием трех упомянутых типов существуют способы достичь почти любого эффекта, который только можно себе вообразить.

Каждый из этих трех типов имеет два ассоциированных с ним элемента, представляющих временные шкалы, с которыми можно работать как с дочерними элементами `Storyboard`. Это следующие шесть элементов: `DoubleAnimation`, `DoubleAnimationUsingKeyFrames`, `PointAnimation`, `PointAnimationUsingKeyFrames`, `ColorAnimation` и `ColorAnimationUsingKeyFrames`. Каждый элемент типа временной шкалы может быть ассоциирован с определенным свойством определенного элемента управления с помощью присоединенных свойств `Storyboard.TargetName` и `Storyboard.TargetProperty`. Например, чтобы выполнить анимацию свойства `Width` элемента управления `Rectangle`, у которого свойство `Name` равно `MyRectangle`, потребуется установить

свойства MyRectangle и Width. Для анимации этого свойства необходимо использовать либо DoubleAnimation, либо DoubleAnimationUsingKeyFrames.

Свойство Storyboard.TargetProperty способно интерпретировать довольно сложный синтаксис. В примере, приведенном в начале этой главы, применялись следующие значения для двух присоединенных свойств:

```
Storyboard.TargetName="ellipse1"
Storyboard.TargetProperty=
"(UIElement.RenderTransform).(TransformGroup.Children)[0]
  .(RotateTransform.Angle)"
```

Элемент ellipse1 имеет тип Ellipse, а свойство TargetProperty указывает угол поворота эллипса посредством анимации. Этот угол находится через свойство RenderTransform элемента Ellipse, унаследованное от UIElement, и первый дочерний элемент объекта TransformGroup, который был значением этого свойства. Первым дочерним элементом был объект RotateTRansform, а угол задан свойством Angle этого объекта.

Хотя этот синтаксис может показаться несколько громоздким, работать с ним очень легко. Сложнее всего определить базовый класс, от которого определенное свойство должно наследоваться, хотя браузер объектов может помочь в этом.

Далее мы рассмотрим простые, не связанные с ключевыми кадрами анимации временные шкалы, после чего перейдем к рассмотрению временных шкал, использующих ключевые кадры.

Временные шкалы без ключевых кадров

Временные шкалы без ключевых кадров — это DoubleAnimation, PointAnimation и ColorAnimation. Эти временные шкалы имеют идентичные имена свойств, хотя типы этих свойств варьируются в зависимости от типа временной шкалы (обратите внимание, что все свойства Duration указываются в коде XAML в форме [дни.]часы:минуты:секунды). Свойства таких временных шкал описаны в табл. 25.2.

Таблица 25.2. Свойства временных шкал без ключевого кадра

Свойство	Описание
Name	Имя временной шкалы, по которому можно ссылаться на нее из других мест
BeginTime	Через какое время инициируется раскадровка после запуска временной шкалы
Duration	Длительность временной шкалы
AutoReverse	Нужен ли по завершении реверс временной шкалы с возвратом свойств в исходные значения. Значение этого свойства имеет булевский тип
RepeatBehavior	Установите это свойство в указанную длительность, чтобы заставить временную шкалу повторяться. Значением может быть целое число, за которым следует x (например, 5x), для повторения временной шкалы указанное количество раз. Значение Forever позволяет бесконечно воспроизводить временную шкалу, пока она не будет остановлена или приостановлена принудительно
FillBehavior	Поведение временной шкалы при ее завершении, но при продолжающейся раскадровке. В качестве значения можно использовать HoldEnd, чтобы оставить свойства в тех значениях, которые у них были на момент завершения временной шкалы (установка по умолчанию), или Stop — для возврата их к исходным значениям

Окончание табл. 25.2

Свойство	Описание
SpeedRatio	Управляет скоростью анимации относительно значений, указанных в других свойствах. Значением по умолчанию является 1, но его можно изменить из другого кода, чтобы ускорить или замедлить анимацию
From	Начальное значение, в которое нужно установить свойство при запуске анимации. Если опустить это значение, будет использоваться текущее значение свойства
To	Финальное значение свойства в конце анимации. Если опустить это значение, будет использоваться текущее значение свойства
By	Используйте это свойство для анимации от текущего значения свойства до суммы текущего значения и заданного здесь значения. Это свойство может использоваться само по себе или в комбинации с From

Например, следующая временная шкала осуществляет анимацию свойства Width элемента управления Rectangle, у которого свойство Name равно MyRectangle, между значениями 100 и 200 за 5 секунд:

```
<Storyboard x:Key="RectangleExpander">
<DoubleAnimation Storyboard.TargetName="MyRectangle"
  Storyboard.TargetProperty="Width" Duration="00:00:05"
  From="100" To="200" />
</Storyboard>
```

Временные шкалы с ключевыми кадрами

Временные шкалы с ключевыми кадрами — это DoubleAnimationUsingKeyFrames, PointAnimationUsingKeyFrames и ColorAnimationUsingKeyFrames. Данные классы временных шкал имеют те же свойства, что и классы временных шкал из предыдущего раздела, за исключением свойств From, To и By. Вместо этого у них есть свойство KeyFrames, представляющее коллекцию объектов — ключевых кадров.

Эти временные шкалы могут содержать любое количество ключевых кадров, каждый из которых может заставить анимируемое значение вести себя иным образом. Существуют три типа ключевых кадров для каждого из типов временной шкалы.

- Discrete. Дискретные ключевые кадры заставляют анимированное значение "перепрыгивать" в указанное значение без перехода.
- Linear. Линейные ключевые кадры заставляют анимируемое значение переходить в указанную величину в процессе линейного перехода.
- Spline. Сплайновые ключевые кадры заставляют анимируемое значение переходить к указанной величине посредством нелинейного перехода, определенного кубической функцией Безье.

Таким образом, существует девять типов объектов ключевых кадров:

- DiscreteDoubleKeyFrame,
- LinearDoubleKeyFrame,
- SplineDoubleKeyFrame,
- DiscreteColorKeyFrame,
- LinearColorKeyFrame,
- SplineColorKeyFrame,

- `DiscretePointKeyFrame`,
- `LinearPointKeyFrame`,
- `SplinePointKeyFrame`.

Классы ключевых кадров обладают теми же тремя свойствами, что и классы временных шкал, рассмотренные в предыдущем разделе, кроме сплайновых ключевых кадров, у которых есть одно дополнительное свойство (табл. 25.3).

Таблица 25.3. Свойства временных шкал с ключевым кадром

Свойство	Описание
`Name`	Имя ключевого кадра, по которому можно ссылаться на него из других мест
`KeyTime`	Местоположение ключевого кадра, выраженное в виде периода времени от момента старта временной шкалы
`Value`	Значение, которого будет достигать свойство, либо которое должно быть установлено по достижению ключевого кадра
`KeySpline`	Два набора из двух чисел в форме `cp1x`, `cp2y`, `cp2x`, `cp2y`, определяющие кубическую функцию Безье, которая используется для анимации свойства. (*Только для сплайновых ключевых кадров.*)

Например, можно выполнить анимацию позиции `Ellipse` в квадрате, анимируя свойство `Center` типа `Paint`, как показано ниже:

```
<Storyboard x:Key="EllipseMover">
  <PointAnimationUsingKeyFrames Storyboard.TargetName="MyEllipse"
    Storyboard.TargetProperty="Center" RepeatBehavior="Forever">
    <LinearPointKeyFrame KeyTime="00:00:00" Value="50,50" />
    <LinearPointKeyFrame KeyTime="00:00:01" Value="100,50" />
    <LinearPointKeyFrame KeyTime="00:00:02" Value="100,100" />
    <LinearPointKeyFrame KeyTime="00:00:03" Value="50,100" />
    <LinearPointKeyFrame KeyTime="00:00:04" Value="50,50" />
  </PointAnimationUsingKeyFrames>
</Storyboard>
```

Значения `Point` в коде XAML задаются в форме `x, y`.

Статические и динамические ресурсы

Другая замечательная особенность WPF заключается в возможности определения ресурсов, таких как стили элементов управления и шаблоны, которые можно использовать многократно по всему приложению. Ресурсы можно даже использовать среди множества приложений, если определить их в правильном месте.

Ресурсы определяются как элементы внутри объекта `ResourceDictionary` — коллекции объектов с ключами. Вот почему до сих пор в примерах настоящей главы во время определения ресурсов применялись атрибуты `x:Key` — чтобы указать ключ, ассоциированный с ресурсом. Обращаться к объектам `ResourceDictionary` можно из самых разных мест. Можно включить ресурсы локально в элемент управления, локально в окно, локально в приложение, или же поместить их во внешнюю сборку.

Для ссылки на ресурсы предусмотрены два способа: статический и динамический. Обратите внимание: это не означает, что сами ресурсы каким-то образом отличаются. То есть ресурс не определяется как статический или динамический; отличие состоит в том, как он используется.

Статические ресурсы

Статические ресурсы применяются, когда во время проектирования в точности известно, каким будет ресурс, и что ссылка на него не изменится на протяжении времени жизни приложения. Например, после определения стиля кнопки, который будет использоваться с кнопками приложения, скорее всего, его не понадобится изменять во время работы приложения. В этом случае к ресурсу следует обращаться статически. Вдобавок, тип статического ресурса определяется во время компиляции, так что их производительность весьма высока.

Для ссылки на статический ресурс применяется следующий синтаксис расширения разметки:

```
{StaticResource имяРесурса}
```

Например, если имеется стиль, определенный для элементов управления `Button` через атрибут `x:Key` стиля `MyStyle`, на него можно сослаться из элемента управления следующим образом:

```
<Button Style="{StaticResource MyStyle}" ...>...</Button>
```

Динамические ресурсы

Динамический ресурс, указанный при определении свойства, во время выполнения может быть заменен другим динамическим ресурсом. Это полезно во многих случаях. Иногда нужно предоставить пользователям контроль над общей темой приложения, и тогда понадобится выделять ресурсы динамически. Кроме того, иногда заранее не известен ключ ресурса, который понадобится во время выполнения, например, при динамическом присоединении сборки ресурсов.

Таким образом, динамические ресурсы обеспечивают более высокую гибкость, чем статические. Однако с ними также связаны и недостатки. Применение динамических ресурсов сопряжено с несколько большими накладными расходами, так что они должны использоваться осторожно, чтобы производительность приложений была оптимальной.

Синтаксис, необходимый для динамической ссылки на ресурсы, очень похож на синтаксис, применяемый при статической ссылке на них:

```
{DynamicResource имяРесурса}
```

Например, если есть стиль, определенный для элементов управления `Button` с атрибутом `x:Key` стиля `MyDynamicStyle`, на него можно ссылаться из элемента управления следующим образом:

```
<Button Style="{DynamicResource MyDynamicStyle}" ...>...</Button>
```

Ссылка на ресурсы стилей

Вы уже видели, как ссылаться на ресурс `Style` из элемента управления `Button`, как статически, так и динамически. Используемый здесь ресурс `Style` может быть в свойстве `Resources` локального элемента управления `Window`, например:

```
<Window ...>
  <Window.Resources>
    <Style x:Key="MyStyle" TargetType="Button">
      ...
    </Style>
  </Window.Resources>
  ...
</Window>
```

Каждый элемент `Button`, который желает получить этот стиль, должен ссылаться на него в свойстве `Style` (статически или динамически). В качестве альтернативы можно

было бы определить статический ресурс как *глобальный* для данного типа элементов управления. То есть объект `Style` применялся бы к *каждому* элементу управления заданного типа, имеющемуся в приложении. Для этого просто опустите атрибут `x:Key`:

```
<Window ...>
  <Window.Resources>
    <Style TargetType="Button">
      ...
    </Style>
  </Window.Resources>
  ...
</Window>
```

Это замечательный способ для установки темы в приложении. Можно определить набор глобальных стилей для различных типов элементов, и эти стили будут применены повсеместно.

В последних нескольких разделах было рассмотрено множество основополагающих аспектов, так что наступило время собрать все вместе в одном примере. В следующем практическом занятии будет модифицирован элемент управления `Button` из предыдущего практического занятия для использования триггеров и анимации, при этом стиль будет определен как глобальный, многократно используемый ресурс.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Триггеры, анимация и ресурсы

1. Создайте приложение WPF по имени `Ch25Ex04` и сохраните его в каталоге `C:\BegVCSharp\Chapter25`.
2. Скопируйте код из файла `MainWindow.xaml` проекта `Ch25Ex03` в одноименный файл проекта `Ch25Ex04`, но измените ссылку на пространство имен в элементе `Window`, как показано ниже:

```
<Window x:Class="Ch25Ex04.MainWindow"
```

*Фрагмент кода **Ch25Ex04\MainWindow.xaml***

3. Скопируйте обработчик событий из файла `MainWindow.xaml.cs` проекта `Ch25Ex03` в файл `MainWindow.xaml.cs` проекта `Ch25Ex04`.
4. Добавьте к элементу `<Window>` дочерний элемент `<Window.Resources>` и переместите в него определение `<Style>` из элемента `<Button.Style>`. Результат (с некоторыми сокращениями) показан ниже:

```
<Window x:Class="Ch25Ex04.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Nasty Button" Height="150" Width="550">
  <Window.Resources>
    <Style TargetType="Button">
      ...
    </Style>
  </Window.Resources>
  <Grid Background="Black">
    <Button Margin="20" Click="Button_Click">
      Would anyone use a button like this?
    </Button>
  </Grid>
</Window>
```

5. Запустите приложение и удостоверьтесь, что получен тот же самый результат, что и в предыдущем примере.

6. Добавьте атрибуты Name к главному элементу управления Grid в шаблоне и элемент Rectangle, содержащий элемент ContentPresenter, как показано ниже:

```
<Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="Button">
      <Grid Name="LayoutGrid">
        <Grid.ColumnDefinitions>
        ...
        <Grid Grid.Column="1">
          <Rectangle RadiusX="10" RadiusY="10" Name="BackgroundRectangle">
            <Rectangle.Fill>
              ...
            </Rectangle.Fill>
          </Rectangle>
          ...
        </Grid>
        ...
      </Grid>
    </ControlTemplate>
  </Setter.Value>
</Setter>
```

7. Добавьте к элементу <ControlTemplate> следующий код, поместив его непосредственно перед дескриптором </ControlTemplate>:

```
  </Grid>
  <ControlTemplate.Resources>
    <Storyboard x:Key="PulseButton">
      <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
        RepeatBehavior="Forever"
        Storyboard.TargetName="BackgroundRectangle"
        Storyboard.TargetProperty=
"(Shape.Fill).(RadialGradientBrush.GradientStops)[1].(GradientStop.Color)">
        <LinearColorKeyFrame Value="Red" KeyTime="00:00:00" />
        <LinearColorKeyFrame Value="Orange" KeyTime="00:00:01" />
        <LinearColorKeyFrame Value="Red" KeyTime="00:00:02" />
      </ColorAnimationUsingKeyFrames>
    </Storyboard>
  </ControlTemplate.Resources>
  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter TargetName="LayoutGrid" Property="Effect">
        <Setter.Value>
          <DropShadowEffect ShadowDepth="0" Color="Red"
            BlurRadius="40" />
        </Setter.Value>
      </Setter>
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter TargetName="LayoutGrid" Property="Effect">
        <Setter.Value>
          <DropShadowEffect ShadowDepth="0" Color="Yellow"
            BlurRadius="80" />
        </Setter.Value>
      </Setter>
    </Trigger>
    <EventTrigger RoutedEvent="UIElement.MouseEnter">
      <BeginStoryboard Storyboard="{StaticResource PulseButton}"
        x:Name="PulseButton_BeginStoryboard" />
    </EventTrigger>
```

```
      <EventTrigger RoutedEvent="UIElement.MouseLeave">
        <StopStoryboard
          BeginStoryboardName="PulseButton_BeginStoryboard" />
      </EventTrigger>
    </ControlTemplate.Triggers>
  </ControlTemplate>
```

8. Запустите приложение и наведите курсор мыши на кнопку. Кнопка начнет пульсировать и светиться (рис. 25.16).

***Рис. 25.16**. Анимация кнопки при наведении на нее курсора мыши*

9. Щелкните на кнопке. Подсветка изменится (рис. 25.17).

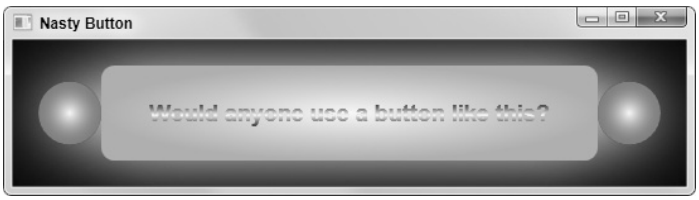


***Рис. 25.17**. Анимация кнопки после щелчка на ней*

Описание работы

В этом примере были сделаны две вещи. Во-первых, определен глобальный ресурс, используемый для форматирования всех кнопок в приложении (хотя в данном случае имеется всего одна кнопка). Во-вторых, добавлены некоторые средства к стилю, созданному в предыдущем практическом занятии. Это позволило заставить кнопку светиться и пульсировать при наведении курсора мыши и после щелчка на ней.

Чтобы сделать стиль глобальным ресурсом, нужно всего лишь перенести элемент `<Style>` в раздел ресурсов `Window`. Можно было бы добавить атрибут `x:Key`, но поскольку это не делалось, то и не было необходимости в установке свойства `Style` элемента управления `Button` на странице; стиль был сделан чисто глобальным.

Сделав стиль ресурсом, мы модифицировали его. Сначала добавили атрибуты `Name` к двум элементам в стиле. Это было необходимо, чтобы можно было обращаться к ним из другого кода, который связан с анимацией и триггерами для шаблона элемента управления, являющегося частью стиля.

Затем мы добавили анимацию в качестве локального ресурса для шаблона элемента управления, указанного в стиле. Объект анимации `Storyboard` был идентифицирован с использованием значения `x:Key` кнопки `PulseButton`:

```
<ControlTemplate.Resources>
  <Storyboard x:Key="PulseButton">
```

Раскадровка содержит элемент `ColorAnimationUsingKeyFRames`, поскольку он осуществляет анимацию цвета, использованного в шаблоне элемента управления. Свойство, подлежащее анимации — красный цвет радиального заполнения, применяемый в элемен-

те управления BackgroundRectangle. Нахождение этого свойства из элемента управления требует довольно сложного синтаксиса для присоединенного свойства Storyboard.TargetProperty:

```
          <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
            RepeatBehavior="Forever"
            Storyboard.TargetName="BackgroundRectangle"
            Storyboard.TargetProperty=
"(Shape.Fill).(RadialGradientBrush.GradientStops)[1].(GradientStop.Color)">
```

Временная шкала анимации состоит из трех ключевых кадров для анимации цвета от Red до Orange и обратно за две секунды:

```
      <LinearColorKeyFrame Value="Red" KeyTime="00:00:00" />
      <LinearColorKeyFrame Value="Orange" KeyTime="00:00:01" />
      <LinearColorKeyFrame Value="Red" KeyTime="00:00:02" />
    </ColorAnimationUsingKeyFrames>
  </Storyboard>
</ControlTemplate.Resources>
```

Добавление анимации как ресурса не означает ее автоматического запуска. Для ее запуска были предусмотрены два триггера EventTrigger:

```
<EventTrigger RoutedEvent="UIElement.MouseEnter">
  <BeginStoryboard Storyboard="{StaticResource PulseButton}"
    x:Name="PulseButton_BeginStoryboard" />
</EventTrigger>
<EventTrigger RoutedEvent="UIElement.MouseLeave">
  <StopStoryboard
    BeginStoryboardName="PulseButton_BeginStoryboard" />
</EventTrigger>
```

В этом коде для управления анимацией используются события MouseEnter и MouseLeave базового класса UIElement кнопки. Событие MouseEnter заставляет анимацию стартовать через элемент BeginSToryboard, а MouseLeave останавливает ее посредством элемента StopSToryboard.

Обратите внимание, что ресурс раскадровки определен локальным в элементе управления, и модифицировать его во время выполнения не планируется.

Также предусмотрены два других триггера для выполнения подсветки при наведении курсора и после щелчка с использованием эффекта DropShadowEffect. Для этого применялись свойства IsMouseOver и IsPressed, показанные ранее в этой главе:

```
<Trigger Property="IsMouseOver" Value="True">
  <Setter TargetName="LayoutGrid" Property="Effect">
    <Setter.Value>
      <DropShadowEffect ShadowDepth="0" Color="Red"
        BlurRadius="40" />
    </Setter.Value>
  </Setter>
</Trigger>
<Trigger Property="IsPressed" Value="True">
  <Setter TargetName="LayoutGrid" Property="Effect">
    <Setter.Value>
      <DropShadowEffect ShadowDepth="0" Color="Yellow"
        BlurRadius="80" />
    </Setter.Value>
  </Setter>
</Trigger>
```

Здесь определена слабая красная подсветка при наведении курсора мыши на кнопку и яркая желтая — при щелчке на ней.

Программирование с использованием WPF

Теперь, ознакомившись со всеми базовыми приемами программирования WPF, можно приступать к созданию собственного приложения. К сожалению, в главе невозможно охватить другие замечательные средства WPF, включая детали привязки данных, а также форматирование при отображении списков. Однако было бы неправильно останавливаться на этом, лишь поверхностно ознакомившись с программированием с помощью WPF. Перед завершением этой главы мы рассмотрим еще две темы, которые выбраны не по причине сложности, а потому, что они отражают задачи, которые, скорее всего, часто придется решать в приложениях WPF:

- создание и использование собственных элементов управления;
- реализация свойств зависимости в собственных элементах управления.

Кроме того, будет рассмотрен последний пример, иллюстрирующий большую часть приемов, изложенных в этой главе, и кратко описана привязка данных WPF.

Пользовательские элементы управления WPF

Технология WPF предоставляет набор элементов управления, полезных во многих ситуациях. Однако, как и все другие платформы для разработки, она также позволяет расширять эту функциональность. В частности, можно создавать собственные элементы управления, наследуя классы от классов из иерархии WPF.

Одним из наиболее полезных классов для наследования элементов управления, является `UserControl`. Этот класс предоставляет всю базовую функциональность, которая, скорее всего, потребуется от элемента управления WPF, в то же время позволяя легко и просто выйти за пределы существующего набора элементов WPF. Все, что достигается с помощью элемента управления WPF — анимация, шаблоны и т.п. — достижимо и с пользовательскими элементами управления.

Для добавления в проект пользовательских элементов управления служит пункт меню Project⇨Add User Control (Проект⇨Добавить пользовательский элемент управления). После этого откроется пустое полотно (canvas), с которым можно работать. Пользовательские элементы управления определяются в XAML с применением элемента `UserControl` высшего уровня, а класс в отделенном коде наследуется от класса `System.Windows.COntrols.UserControl`.

После добавления в проект пользовательского элемента управления можно включать в него элементы, создавая его компоновку, и отделенный код — для его конфигурирования. Завершенный новый элемент можно использовать по всему приложению и даже повторно использовать в других приложениях.

Одной из важнейших вещей, которые следует знать при создании собственных элементов управления, является реализация свойств зависимости. Как было показано ранее в этой главе, свойства зависимости — важнейшая часть программирования WPF. При создании собственных элементов управления вы не обойдетесь без функциональности, обеспечиваемой этими свойствами.

Реализация свойств зависимости

Свойства зависимости могут быть добавлены к любому классу, унаследованному от `System.Windows.DependencyObject`. Этот класс присутствует в иерархии наследования многих классов WPF, включая все элементы управления и `UserControl`.

Чтобы реализовать свойство зависимости в классе, к его определению добавляется общедоступный статический член типа `System.Windows.DependencyProperty`. Имя этого

члена может быть произвольным, но обычно стоит следовать соглашению об именовании вида *<ИмяСвойства>*Property:

```
public static DependencyProperty MyStringProperty;
```

Может показаться странным, что это свойство определено как статическое, поскольку в конечном итоге требуется свойство, которое может быть уникально идентифицировано для каждого экземпляра класса. Система свойств WPF отслеживает эти вещи самостоятельно, так что пока об этом беспокоиться не нужно.

Добавляемый член должен быть сконфигурирован с использованием статического метода DependencyProperty.Register():

```
public static DependencyProperty MyStringProperty =
  DependencyProperty.Register(...);
```

Этот метод принимает от трех до пяти параметров, которые описаны в табл. 25.4 (первые три из пяти параметров являются обязательными).

Таблица 25.4. Параметры метода `DependencyProperty.Register()`

Параметр	Описание
`string name`	Имя свойства
`Type propertyType`	Тип свойства
`Type ownerType`	Тип класса, содержащего свойство
`PropertyMetadata typeMetadata`	Дополнительные настройки свойства: значение свойства по умолчанию и методы обратного вызова для использования в уведомлениях об изменении свойства и приведении
`ValidateValueCallback validateValueCallback`	Метод обратного вызова, используемый для проверки достоверности значений свойства

Существуют и другие методы, которые можно использовать для регистрации свойств зависимости, такие как `RegisterAttached()`, служащий для реализации присоединенного свойства. Эти методы в настоящей главе не рассматриваются, однако стоит о них почитать самостоятельно.

Например, можно было бы зарегистрировать свойство зависимости MyStringProperty, используя три параметра, как показано ниже:

```
public class MyClass : DependencyObject
{
  public static DependencyProperty MyStringProperty =
     DependencyProperty.Register(
    "MyString",
    typeof(string),
    typeof(MyClass));
}
```

Можно также включить свойство .NET, используемое для доступа к свойствам зависимости напрямую (хотя это и не обязательно, как вы вскоре убедитесь). Однако, поскольку свойства зависимости определены как статические члены, применять к ним тот же самый синтаксис, что и для обычных свойств, нельзя. Чтобы доступа к значению свойства зависимости должны использоваться методы, унаследованные от DependencyObject:

```
public class MyClass : DependencyObject
{
  public static DependencyProperty MyStringProperty = DependencyProperty.Register(
    "MyString", typeof(string), typeof(MyClass));
  public string MyString
  {
    get { return (string)GetValue(MyStringProperty); }
    set { SetValue(MyStringProperty, value); }
  }
}
```

Здесь методы `GetValue()` и `SetValue()`, соответственно, получают и устанавливают значение свойства зависимости `MyStringProperty` для текущего экземпляра. Эти два метода общедоступны, так что клиентский код может использовать их непосредственно для манипуляции значениями свойства зависимости. Именно поэтому добавлять свойство .NET для доступа к свойству зависимости не обязательно.

Чтобы установить метаданные для свойства, необходимо воспользоваться объектом, унаследованным от `PropertyMetadata`, таким как `FrameworkPropertyMetadata`, и передавать этот экземпляр методу `Register()` в четвертом параметре. Существует 11 перегрузок конструктора `FrameworkPropertyMetadata`, и они принимают один или более параметров, перечисленных в табл. 25.5.

Таблица 25.5. Параметры конструкторов `FrameworkPropertyMetadata`

Тип параметра	Описание
`object defaultValue`	Значение свойства по умолчанию
`FrameworkPropertyMetadataOptions flags`	Комбинация флагов (из перечисления `FrameworkPropertyMetadataOptions`), позволяющих указать дополнительные метаданные для свойства. Например, с помощью флага `AffectsArrange` объявляется, что изменения свойства могут затронуть компоновку элемента управления. Это заставит механизм компоновки окна заново вычислять компоновку при изменении свойства. Полный список доступных флагов приведен в документации MSDN
`PropertyChangedCallback propertyChangedCallback`	Метод обратного вызова для использования при изменении значения свойства
`CoerceValueCallback coerceValueCallback`	Метод обратного вызова для использования при выполнении приведения значения свойства
`bool isAnimationProhibited`	Указывает, может ли данное свойство изменяться с помощью анимации
`UpdateSourceTrigger defaultUpdateSourceTrigger`	Когда значения свойств привязаны к данным, это свойство определяет, когда должны изменяться исходные данные, в соответствии со значениями из перечисления `UpdateSourceTrigger`. Значением по умолчанию является `PropertyChanged`, которое означает, что привязываемое свойство обновляется немедленно при изменении свойства. Это не всегда то, что нужно. Например, свойство `TextBox.Text` использует значение `LostFocus` для этого свойства. Это гарантирует, что привязанный источник не будет обновлен преждевременно. Можно также применять значение `Explicit`, указывая, что привязанный источник должен обновляться по запросу (вызовом метода `UpdateSource()` класса, унаследованного от `DependencyObject`)

Ниже показан простой пример использования FrameworkPropertyMetadata для установки значения свойства по умолчанию:

```
public class MyClass : DependencyObject
{
  public static DependencyProperty MyStringProperty =
       DependencyProperty.Register(
       "MyString",
       typeof(string),
       typeof(MyClass),
       new FrameworkPropertyMetadata("Default value"));
}
```

Ранее упоминались три метода обратного вызова, которые можно задавать для уведомления об изменении свойства, для приведения свойства и для проверки достоверности значения свойства. Эти обратные вызовы, подобно самому свойству зависимости, должны быть реализованы как общедоступные статические методы. Каждый обратный вызов имеет специфический тип возврата и список параметров.

В следующем практическом занятии будет создан пользовательский элемент управления с двумя свойствами зависимости. Вы увидите, как реализовать методы обратного вызова для этих свойств в коде пользовательского элемента управления.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Пользовательские элементы управления

1. Создайте приложение WPF по имени Ch25Ex05 и сохраните его в каталоге C:\BegVCSharp\Chapter25.

2. Добавьте к приложению новый пользовательский элемент управления по имени Card и модифицируйте код в файле Card.xaml, как показано ниже:

```
<UserControl x:Class="Ch25Ex05.Card"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  mc:Ignorable="d"
  d:DesignHeight="150" d:DesignWidth="100"
  Height="150" Width="100" x:Name="UserControl"
  FontSize="16" FontWeight="Bold">
  <UserControl.Resources>
    <DataTemplate x:Key="SuitTemplate">
      <TextBlock Text="{Binding}"/>
    </DataTemplate>
  </UserControl.Resources>
  <Grid>
    <Rectangle Stroke="{x:Null}" RadiusX="12.5" RadiusY="12.5">
      <Rectangle.Fill>
        <LinearGradientBrush EndPoint="0.47,-0.167" StartPoint="0.86,0.92">
          <GradientStop Color="#FFD1C78F" Offset="0"/>
          <GradientStop Color="#FFFFFFFF" Offset="1"/>
        </LinearGradientBrush>
      </Rectangle.Fill>
      <Rectangle.Effect>
        <DropShadowEffect/>
      </Rectangle.Effect>
    </Rectangle>
    <Label x:Name="SuitLabel"
    Content="{Binding Path=Suit, ElementName=UserControl, Mode=Default}"
      ContentTemplate="{DynamicResource SuitTemplate}"
      HorizontalAlignment="Center" VerticalAlignment="Center"
```

```
      Margin="8,51,8,60" />
    <Label x:Name="RankLabel"
      Content="{Binding Path=Rank, ElementName=UserControl, Mode=Default}"
      ContentTemplate="{DynamicResource SuitTemplate}"
      HorizontalAlignment="Left" VerticalAlignment="Top"
      Margin="8,8,0,0" />
    <Label x:Name="RankLabelInverted"
      Content="{Binding Path=Rank, ElementName=UserControl, Mode=Default}"
      ContentTemplate="{DynamicResource SuitTemplate}"
      HorizontalAlignment="Right" VerticalAlignment="Bottom"
      Margin="0,0,8,8" RenderTransformOrigin="0.5,0.5">
      <Label.RenderTransform>
        <RotateTransform Angle="180"/>
      </Label.RenderTransform>
    </Label>
    <Path Fill="#FFFFFFFF" Stretch="Fill" Stroke="{x:Null}"
      Margin="0,0,35.218,-0.077" Data="F1 M110.5,51 L123.16457,51 C116.5986,
        76.731148 115.63518,132.69684 121.63533,149.34013 133.45299,
        182.12018 152.15821,195.69803 161.79765,200.07669 L110.5,200 C103.59644,
        200 98,194.40356 98,187.5 L98,63.5 C98,56.596439 103.59644,51 110.5,51 z">
      <Path.OpacityMask>
        <LinearGradientBrush EndPoint="0.957,1.127" StartPoint="0,-0.06">
          <GradientStop Color="#FF000000" Offset="0"/>
          <GradientStop Color="#00FFFFFF" Offset="1"/>
        </LinearGradientBrush>
      </Path.OpacityMask>
    </Path>
  </Grid>
</UserControl>
```

Фрагмент кода ***Ch25Ex05\Card.xaml***

3. Измените код в `Card.xaml.cs` следующим образом:

```
public partial class Card : UserControl
{
  public static string[] Suits = { "Club", "Diamond", "Heart", "Spade" };
  public static DependencyProperty SuitProperty = DependencyProperty.Register(
    "Suit",
    typeof(string),
    typeof(Card),
    new PropertyMetadata("Club", new PropertyChangedCallback(OnSuitChanged)),
    new ValidateValueCallback(ValidateSuit));
  public static DependencyProperty RankProperty = DependencyProperty.Register(
    "Rank",
    typeof(int),
    typeof(Card),
    new PropertyMetadata(1),
    new ValidateValueCallback(ValidateRank));
  public Card()
  {
    InitializeComponent();
  }
  public string Suit
  {
    get { return (string)GetValue(SuitProperty); }
    set { SetValue(SuitProperty, value); }
  }
  public int Rank
  {
    get { return (int)GetValue(RankProperty); }
    set { SetValue(RankProperty, value); }
  }
```

```
  public static bool ValidateSuit(object suitValue)
  {
    string suitValueString = (string)suitValue;
    if (suitValueString != "Club" && suitValueString != "Diamond"
        && suitValueString != "Heart" && suitValueString != "Spade")
    {
      return false;
    }
    return true;
  }
  public static bool ValidateRank(object rankValue)
  {
    int rankValueInt = (int)rankValue;
    if (rankValueInt < 1 || rankValueInt > 12)
    {
      return false;
    }
    return true;
  }
  private void SetTextColor()
  {
    if (Suit == "Club" || Suit == "Spade")
    {
      RankLabel.Foreground = new SolidColorBrush(Color.FromRgb(0, 0, 0));
      SuitLabel.Foreground = new SolidColorBrush(Color.FromRgb(0, 0, 0));
      RankLabelInverted.Foreground =
        new SolidColorBrush(Color.FromRgb(0, 0, 0));
    }
    else
    {
      RankLabel.Foreground = new SolidColorBrush(Color.FromRgb(255, 0, 0));
      SuitLabel.Foreground = new SolidColorBrush(Color.FromRgb(255, 0, 0));
      RankLabelInverted.Foreground =
        new SolidColorBrush(Color.FromRgb(255, 0, 0));
    }
  }
  public static void OnSuitChanged(DependencyObject source,
    DependencyPropertyChangedEventArgs args)
  {
    ((Card)source).SetTextColor();
  }
}
```

Фрагмент кода ***Ch25Ex05\Card.xaml.cs***

4. Модифицируйте код `MainWindow.xaml`, как показано ниже:

```
<Window x:Class="Ch25Ex05.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Card Dealer" Height="600" Width="800">
  <Grid Name="contentGrid" MouseLeftButtonDown="Grid_MouseLeftButtonDown"
    MouseLeftButtonUp="Grid_MouseLeftButtonUp" MouseMove="Grid_MouseMove">
    <Grid.Background>
      <LinearGradientBrush EndPoint="0.364,0.128" StartPoint="0.598,1.042">
        <GradientStop Color="#FF0D4F1A" Offset="0"/>
        <GradientStop Color="#FF448251" Offset="1"/>
      </LinearGradientBrush>
    </Grid.Background>
  </Grid>
</Window>
```

Фрагмент кода ***Ch25Ex05\MainWindow.xaml***

5. Измените код MainWindow.xaml.cs следующим образом:

```
public partial class MainWindow : Window
{
  private Card currentCard;
  private Point offset;
  private Random random = new Random();
  public MainWindow()
  {
    InitializeComponent();
  }
  private void Grid_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
  {
    if (e.Source is Card)
    {
      currentCard = (Card)e.Source;
      offset = Mouse.GetPosition(currentCard);
      contentGrid.Children.Remove(currentCard);
    }
    else
    {
      currentCard = new Card
      {
        Suit = Card.Suits[random.Next(0, 4)],
        Rank = random.Next(1, 13)
      };
      currentCard.HorizontalAlignment = HorizontalAlignment.Left;
      currentCard.VerticalAlignment = VerticalAlignment.Top;
      offset = new Point(50, 75);
    }
    contentGrid.Children.Add(currentCard);
    PositionCard();
  }
  private void Grid_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
  {
    currentCard = null;
  }
  private void Grid_MouseMove(object sender, MouseEventArgs e)
  {
    if (currentCard != null)
    {
      PositionCard();
    }
  }
  private void PositionCard()
  {
    Point mousePos = Mouse.GetPosition(this);
    currentCard.Margin =
      new Thickness(mousePos.X - offset.X, mousePos.Y - offset.Y, 0, 0);
  }
}
```

Фрагмент кода ***Ch25Ex05\MainWindow.xaml.cs***

6. Запустите приложение. Щелкните на поверхности окна для добавления случайных карт, затем щелкайте и перетаскивайте соответствующие карты. Щелчок на имеющейся карте вызывает ее перемещение на вершину в порядке стека. Результат показан на рис. 25.18.

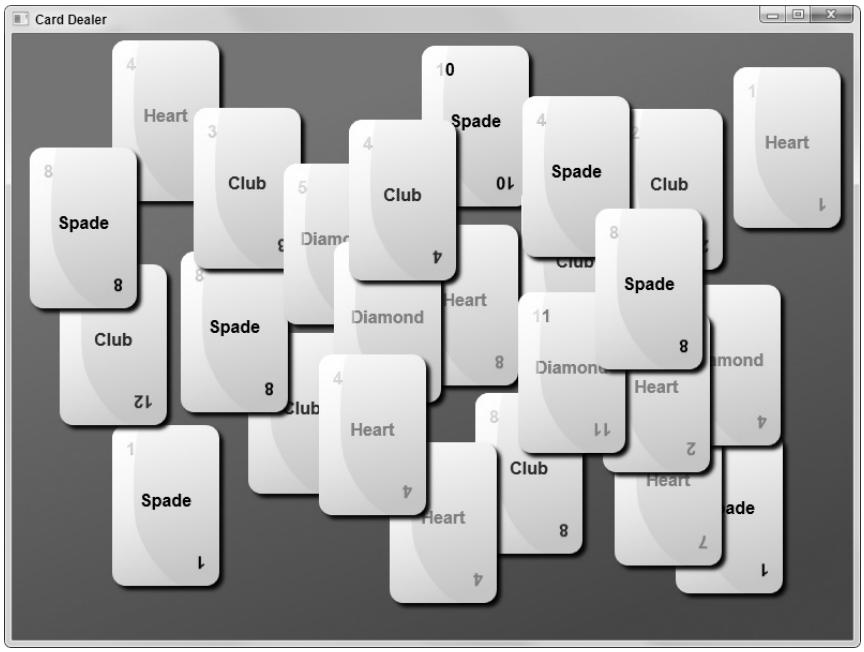


***Рис. 25.18**. Приложение `Ch25Ex05` в работе*

Описание работы

В этом примере создается пользовательский элемент управления с двумя свойствами зависимости, а также клиентский код для работы с ним. Пример охватывает большую часть основ и его рассмотрение стоит начать с кода элемента управления `Card`.

Элемент управления `Card` состоит в основном из кода, который должен быть знаком, т.к. приводился в этой главе ранее. Код компоновки не содержит ничего нового, хотя нельзя не согласиться с тем, что полученный результат более впечатляет, чем кнопка в предыдущих двух примерах. Новым в этом примере является наличие небольшого объема кода *привязки данных*. Привязка обеспечивает связь элемента управления с источником данных, и как таковая, обеспечивает широкий диапазон технических приемов. WPF позволяет легко привязывать свойства к различного рода источникам данных, таким как базы данных, документы XML и (как в рассматриваемом примере) значения свойств зависимости.

В частности, код `Card` представляет клиентскому коду два свойства зависимости — `Suit` и `Rank`, и привязывает эти свойства к визуальным элементам в данной компоновке элементов управления. В результате, когда в `Suit` устанавливается значение `Club`, слово "Club" отображается по центру карты. Аналогично значение `Rank` отображается в двух углах карты.

Скоро мы рассмотрим реализацию `Suit` и `Rank`. А пока достаточно знать, что эти свойства хранят значения, соответственно, `string` и `int`. Можно было бы хранить в них значения перечислений, но это потребовало бы немного больше кода, а мы старались сделать этот пример насколько возможно, простым, используя лишь базовые типы.

Для привязки значения к свойству применяется синтаксис привязки, который является расширением разметки. Этот синтаксис означает, что значение свойства указывается как `{Binding...}`. Существуют разные способы конфигурирования привязки. В данном примере привязка метки `SuitLabel` сконфигурирована следующим образом:

```
<Label x:Name="SuitLabel"
  Content="{Binding Path=Suit, ElementName=UserControl, Mode=Default}"
  ContentTemplate="{DynamicResource SuitTemplate}" HorizontalAlignment="Center"
  VerticalAlignment="Center" Margin="8,51,8,60" />
```

Здесь указаны три свойства для привязки: `Path` (имя свойства), `ElementName` (элемент со свойством) и `Mode` (как выполнять привязку). Свойства `Path` и `Element` достаточно очевидны, а `Mode` пока можно проигнорировать. Важно отметить, что эта спецификация привязывает свойство `Label.Content` к свойству `Card.Suit`.

Во время привязки значений свойств должно быть также указано, как следует визуализировать привязанное содержимое, с использованием *шаблона данных*. В рассматриваемом примере шаблоном данных является `SuitTemplate`, на который производится ссылка как на динамический ресурс (хотя здесь прекрасно бы работал и статический ресурс). Этот шаблон определен в разделе ресурсов пользовательского элемента управления:

```
<UserControl.Resources>
  <DataTemplate x:Key="SuitTemplate">
    <TextBlock Text="{Binding}"/>
  </DataTemplate>
</UserControl.Resources>
```

Таким образом, строковое значение `Suit` используется как свойство `Text` элемента управления `TextBlock`. То же определение `DataTemplate` применяется и для двух меток ранга — неважно, что типом `Rank` является `int`, он все равно преобразуется в `string`, когда привязывается к свойству `TextBlock.Text`.

Очевидно, что о привязке данных и шаблонах данных можно было бы рассказать намного больше, однако в этой книге просто нет места, чтобы углубиться во все детали. Ниже в резюме будут даны источники, откуда можно почерпнуть дополнительную информацию по этой теме. В качестве завершающего примечания следует отметить, что если вы пользуетесь Expression Blend, то обнаружите, что можно эффективно привязывать данные, не особенно вдаваясь в детали синтаксиса XAML, поскольку инструмент об этом заботится сам.

Для работы этой привязки данных должны быть определены два свойства зависимости с помощью приемов, описанных в предыдущем разделе. Они определены в отделенном коде для пользовательского элемента управления, как показано ниже (оба они имеют простые свойства-оболочки .NET, которые нет необходимости показывать здесь ввиду простоты кода):

```
public static DependencyProperty SuitProperty =
   DependencyProperty.Register(
      "Suit",
      typeof(string),
      typeof(Card),
      new PropertyMetadata(
      "Club", new PropertyChangedCallback(OnSuitChanged)),
      new ValidateValueCallback(ValidateSuit));
public static DependencyProperty RankProperty =
      DependencyProperty.Register(
      "Rank",
      typeof(int),
      typeof(Card),
      new PropertyMetadata(1),
      new ValidateValueCallback(ValidateRank));
```

Оба свойства зависимости используют метод обратного вызова для проверки достоверности значений, а свойство `Suit` также имеет метод обратного вызова на случай из-

менения его значения. Методы обратного вызова для проверки достоверности имеют тип возврата bool и единственный параметр типа object, представляющий собой значение, которое клиентский код пытается установить в свойстве. Если значение в порядке, следует вернуть true, а иначе — false. В этом примере кода свойство Suit ограничено одной из четырех строк:

```
public static bool ValidateSuit(object suitValue)
{
  string suitValueString = (string)suitValue;
  if (suitValueString != "Club" && suitValueString != "Diamond"
     && suitValueString != "Heart" && suitValueString != "Spade")
  {
    return false;
  }
  return true;
}
```

Это довольно грубый подход, и очевидно, что здесь бы лучше подошло перечисление, но мы избегаем его по причинам, изложенным выше. Аналогично, свойство Rank ограничено значениями от 1 (туз) до 12 (король):

```
public static bool ValidateRank(object rankValue)
{
  int rankValueInt = (int)rankValue;
  if (rankValueInt < 1 || rankValueInt > 12)
  {
    return false;
  }
  return true;
}
```

Когда изменяется значение Suit, происходит обращение к методу обратного вызова OnSuitChanged(). Этот метод отвечает за установку красного (для червей и бубен) или черного (для трефов и пик) цвета текста. Это делается посредством вызова служебного метода на источнике вызова метода. Причина в том, что метод обратного вызова реализован как статический, но он принимает в качестве параметра экземпляр пользовательского элемента управления, который инициировал событие, так что может взаимодействовать с ним. Вызываемый служебный метод имеет имя SetTextColor():

```
public static void OnSuitChanged(DependencyObject source,
DependencyPropertyChangedEventArgs args)
{
  ((Card)source).SetTextColor();
}
```

Метод SetTextColor() является приватным, но, тем не менее, он доступен в OnSuitChanged(), поскольку оба они — члены одного класса, несмотря на то, что один является методом экземпляра, а другой — статическим методом. SetTextColor() просто устанавливает свойство Foreground различных меток элемента управления в кисть сплошного цвета, которая может быть либо красного, либо черного цвета, в зависимости от значения Suit:

```
private void SetTextColor()
{
  if (Suit == "Club" || Suit == "Spade")
  {
    RankLabel.Foreground =
      new SolidColorBrush(Color.FromRgb(0, 0, 0));
    SuitLabel.Foreground =
      new SolidColorBrush(Color.FromRgb(0, 0, 0));
```

```
      RankLabelInverted.Foreground =
        new SolidColorBrush(Color.FromRgb(0, 0, 0));
    }
    else
    {
      RankLabel.Foreground =
        new SolidColorBrush(Color.FromRgb(255, 0, 0));
      SuitLabel.Foreground =
        new SolidColorBrush(Color.FromRgb(255, 0, 0));
      RankLabelInverted.Foreground =
        new SolidColorBrush(Color.FromRgb(255, 0, 0));
    }
  }
```

Это все, на что нужно обратить внимание в коде элемента управления `Card`. Клиентский код в `MainWindow.xaml` и `MainWindow.xaml.cs` достаточно прост. В нем используется некоторая базовая стилизация для обеспечения градиентного зеленого фона, а также набор обработчиков событий (с помощью маршрутизируемых и присоединенных событий) для организации взаимодействия с пользователем. Здесь присутствует еще пара трюков, например, применение полей для позиционирования карт и смещения, позволяющего перетаскивать карты за точку, на которой был совершен щелчок, но ничего такого, в чем не удастся разобраться при самостоятельном изучении кода.

Резюме

В этой главе вы узнали все, что необходимо знать для начала программирования с использованием WPF. Вы ознакомились, хотя и кратко, с некоторыми более сложными приемами, которые должны были дать почувствовать вкус того, что позволяет расширенное программирование WPF. Технология WPF — тема слишком обширная, чтобы полностью охватить ее в одной главе, и если она заинтересовала вас, следует обратиться к дополнительным источникам. Если интересуют детали XAML в веб-среде, начать стоит с руководства *WPF: Windows Presentation Foundation в .NET 4.0 с примерами на C# 2010 для профессионалов* (ИД "Вильямс", 2011 г.) или *Silverlight 3 с примерами на C# для профессионалов* (ИД "Вильямс", 2010 г.). Silverlight — особенно впечатляющая область на данный момент, и ее возможности постоянно совершенствуются. Учитывая, что Silverlight, по сути — подмножество WPF для веб-разработки, важно отметить, что навыки работы с WPF и Silverlight являются переносимыми.

Разумеется, имеет смысл опробовать доступные инструменты, в частности, Expression Blend, и увидеть, чего с их помощью можно достичь. Документация MSDN также окажет помощь, хотя из-за того, что WPF — все еще новая технология, в этом ресурсе вы столкнетесь с немалым числом пробелов, которые понадобится компенсировать другими источниками.

Существует ряд замечательных веб-сайтов, где можно поискать дополнительную информацию. В частности, обратитесь на сайт сообщества WPF по адресу `http://windowsclient.net`, а также не упускайте из виду блог Скотта Гатри (Scott Guthrie) на `http://weblogs.asp.net/scottgu`.

В этой главе рассматривались следующие темы.

- Что собой представляет технология WPF и ее потенциальное влияние на разработку как настольных, так и веб-приложений.
- Как архитектура WPF позволяет дизайнерам и разработчикам работать вместе в проектах, используя и Expression Blend, и VS или VCE.
- Что такое язык XAML, как работает базовый синтаксис XAML, и что означает некоторая связанная с ним терминология.

- Как используется объект `Application`.
- Как работают элементы управления WPF, включая концепции свойств зависимости и присоединенных свойств, а также маршрутизируемых и присоединенных событий.
- Как работает система компоновки WPF, и как использовать различные контейнеры компоновки для позиционирования элементов.
- Как применять стили и шаблоны для настройки внешнего вида и поведения элементов управления.
- Как использовать триггеры и анимации для улучшения впечатления пользователя от приложений.
- Как определять ресурсы во внутренних и внешних словарях ресурсов, и как обращаться к ресурсам статически и динамически.
- Как создавать пользовательские элементы управления со свойствами зависимости.

Следующая глава посвящена другой технологии, которая появилась в .NET 3.0: Windows Communication Foundation.

Упражнения

1. Для настольных приложений WPF и браузерных приложений WPF можно использовать в точности одинаковый код XAML. Верно ли это утверждение?
2. Какой техникой необходимо воспользоваться, чтобы позволить дочерним элементам управления устанавливать индивидуальные значения для свойства, определенного в родительском элементе? Какой синтаксис должен применяться в XAML для достижения этого? Приведите пример кода XAML, в котором два дочерних элемента управления `Branch` устанавливают разные значения для свойства `LeafCount`, определенного в родительском элементе управления `Tree`.
3. Какие из следующих утверждений о свойствах зависимости верны?
   а) Свойства зависимости должны быть доступны через ассоциированное свойство .NET.
   б) Свойства зависимости определяются как общедоступные статические члены.
   в) Допускается иметь только одно свойство зависимости на определение класса.
   г) Свойства зависимости должны называться с использованием соглашения о наименовании *<ИмяСвойства>*`Property`.
   д) Можно проверять достоверность значений, присваиваемых свойству зависимости, с помощью метода обратного вызова.
4. Какой элемент управления компоновкой должен использоваться для отображения элементов управления в одной строке или столбце?
5. Туннелируемые события в WPF именуются определенным образом, чтобы их можно было идентифицировать. Что это за соглашение об именовании?
6. Для каких типов свойств можно выполнять анимацию?
7. Когда должны использоваться динамические ссылки на ресурсы вместо статических ссылок?

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Основные концепции
Что собой представляет WPF	Технология WPF — это относительно недавно появившаяся методика от Microsoft, которая применяется для создания Windows- и веб-приложений. В ней используется модель с разметкой и отделенным кодом, которая обеспечивает четкое разделение между дизайном и функциональностью, что делает возможной одновременную работу над проектом дизайнеров и разработчиков. Разметка в WPF производится с помощью языка XAML. Для работы XAML дизайнеры часто используют инструмент Expression Blend.
XAML	Синтаксис XAML позволяет представлять объекты в разметке. Язык XAML построен на основе XML, а для предоставления средств WPF в нем используются расширения разметки.
Элементы управления	С помощью элементов управления WPF можно строить пользовательские интерфейсы, подобные Windows Forms. Элементы управления WPF используют свойства зависимости для интеграции с разнообразными аспектами платформы, такими как уведомление об изменениях. Свойства зависимости могут быть присоединены к объектам, отличным от тех, в которых они определены, предоставляя, где требуется, контекстную информацию. Событиями элементов управления в WPF обычно являются маршрутизируемые события, которые туннелируются и распространяются подобно пузырьку по всей иерархии элементов управления в WPF-приложении.
Компоновка	WPF поставляется с множеством элементов управления компоновкой, которые могут содержать в себе другие элементы управления. В зависимости от желаемого типа компоновки, можно использовать такие элементы управления компоновкой, как `Canvas`, `DockPanel`, `Grid`, `StackPanel` или `WrapPanel`.
Стилизация	Стилизация элементов управления осуществляется с помощью объектов `Style`, которые в основном состоят из объектов `Setter`, применяемых к свойствам элементов управления. Изменяя шаблон, можно полностью управлять внешним видом и поведением элемента управления. С помощью триггеров можно реагировать на пользовательское взаимодействие и изменения в данных.
Анимация	Над свойствами можно выполнять анимацию для диапазона значений и временного промежутка. С помощью ключевых кадров можно определить существенные точки в анимации, после чего запускать и останавливать ее программно или посредством триггеров.
Ресурсы	Ресурсы, в частности, стили и шаблоны, могут быть определены на любом уровне — например, элемента управления, окна или приложения. Доступ к ресурсам осуществляется с помощью расширений разметки `StaticResource` и `DynamicResource`, что зависит от того, нужно ли изменять ресурс во время выполнения.

26

Windows Communication Foundation

В ЭТОЙ ГЛАВЕ...

- Что собой представляет WCF
- Концепции WCF
- Программирование с использованием WCF

В главе 19 вы узнали о веб-службах и о том, как использовать их для обеспечения простых коммуникаций между приложениями. Было показано, как применять HTTP-запросы `GET` и `POST` для обмена данными с веб-службами и как работать с протоколом SOAP. За годы, прошедшие с момента, когда веб-службы впервые стали доступны разработчикам .NET, стало ясно, что хотя веб-службы и великолепны, этой технологии еще есть куда развиваться. В связи с этим в Microsoft выпустили дополнение WSE (Web Services Enhancements — расширения веб-служб). WSE позволяет разработчикам веб-служб включать средства безопасности сообщений, предусматривает приемы маршрутизации, а также различные прочие политики для усовершенствования веб-служб. Однако и здесь еще есть место для улучшений.

Другая технология .NET — Remoting — позволяет создавать экземпляры объектов в одном процессе и использовать их в другом. Это возможно, даже если объект создан на другом компьютере, а не на том, где он используется. Однако у этой технологии все еще есть свои проблемы — она ограничена, и ею не так просто овладеть начинающим программистам.

Технология Windows Communication Foundation (WCF) — это, по сути, замена как веб-служб, так и Remoting. Она заимствует такие концепции, как службы и независимый от платформы обмен сообщениями, от веб-служб, комбинируя их с концепциями Remoting, среди которых серверные хост-приложения и расширенные возможности привязки. В результате получается технология, которую можно воспринимать как надмножество, включающее в себя и веб-службы, и Remoting, но гораздо более мощную, чем веб-службы и более простую в применении, чем Remoting. Благодаря WCF, можно перейти от простых приложений к приложениям, использующим *архитектуру, ориентированную на службы* (service-oriented architecture — SOA). Архитектура SOA означает децентрализацию и распределение обработки, когда подключение к службам и данным осуществляется по мере необходимости через локальные сети и Интернет.

В этой главе будут рассмотрены принципы, лежащие в основе WCF, и показано, как создавать и потреблять службы WCF в коде приложений.

Создавать службы WCF в Visual C# 2010 Express (VCE) нельзя, для этого должна использоваться полная версия Visual Studio 2010 (VS). Создавать располагающиеся на сервере IIS службы WCF можно также в среде Visual Web Developer 2010 Express, но в этой главе будет использоваться VS, чтобы продемонстрировать полный набор опций.

Что собой представляет WCF

WCF — это технология, позволяющая создавать службы, к которым можно обращаться из других приложений через границы процессов, машин и сетей. Эти службы можно использовать для разделения функциональности между множеством приложений, предоставления источников данных или абстрагирования сложных процессов.

Как и в веб-службах, функциональность, обеспечиваемая службами WCF, инкапсулирована в индивидуальных методах, предоставляемых службой. Каждый метод, или в терминологии WCF — *операция*, имеет конечную точку, с которой производится обмен данными для ее использования. В этом WCF отличается от веб-служб. В веб-службах с конечной точкой допускается взаимодействовать только по протоколу SOAP поверх HTTP. Службы WCF предлагают выбор необходимого протокола. Можно даже иметь конечные точки, которые взаимодействуют по более чем одному протоколу, в зависимости от сети, через которую осуществляется подключение к службе, и специфических требований.

В WCF конечная точка может иметь несколько *привязок*, каждая из которых указывает средство коммуникации. Привязки могут также специфицировать дополнительную информацию, такую как требования безопасности, которые должны соблюдаться для коммуникаций с конечной точкой. Привязка может требовать аутентификацию с помощью имени

пользователя и пароля либо маркер регистрационной записи пользователя Windows. При подключении к конечной точке протокол, используемый привязкой, влияет на адрес, по которому производится обращение, как будет показано ниже.

После подключения с конечной точкой можно взаимодействовать посредством сообщений SOAP. Используемая форма сообщений зависит от операции и структур данных, которые должны быть отправлены или получены в сообщении в рамках этой операции. Для спецификации всего этого в WCF применяются *контракты*. Контракты можно исследовать через метаданные, полученные от службы. Это аналог применения языка WSDL для описания функциональности веб-служб. Фактически информацию о службе WCF можно получать в формате WSDL, хотя службы WCF могут быть описаны также и другими способами.

После идентификации службы и конечной точки и выяснения, какая привязка применяется, и какому контракту она соответствует, со службой WCF можно взаимодействовать так же легко, как с определенным локально объектом. Коммуникации со службами WCF могут быть простыми, однонаправленными транзакциями, сообщениями "запрос/ответ" либо полнодуплексными коммуникациями, инициированными на любом конце коммуникационного канала. Допускается также использовать приемы оптимизации рабочей нагрузки, такие как MTOM (Message Transmission Optimization Mechanism — механизм оптимизации передачи сообщений), чтобы при необходимости упаковывать данные.

Сама служба WCF при развертывании может запускаться в одном из процессов на компьютере, где она размещается. В отличие от веб-служб, которые всегда выполняются на сервере IIS, для служб WCF можно выбирать хост-процесс, подходящий в конкретной ситуации. В качестве хоста для служб WCF разрешено использовать не только IIS, но также службы Windows и исполняемые программы. Если для взаимодействия со службой WCF по локальной сети применяется протокол TCP, то в этом случае даже нет необходимости в установке IIS на компьютере, служащем хостом для службы.

Технология WCF спроектирована так, что позволяет настраивать почти все, что было описано в данном разделе. Однако это сложная тема, и в настоящей главе мы ограничимся рассмотрением только приемов, предоставляемых .NET 4 по умолчанию.

После ознакомления с основами служб WCF в последующих разделах будут подробно описаны связанные с ними детали.

Концепции WCF

В этом разделе рассматриваются следующие аспекты WCF:

- коммуникационные протоколы WCF;
- адреса, конечные точки и привязки;
- контракты;
- шаблоны сообщений;
- поведения;
- хостинг.

Коммуникационные протоколы WCF

Как было описано ранее, со службами WCF можно взаимодействовать посредством разнообразных транспортных протоколов. Фактически в .NET 4 Framework определено четыре таких протокола.

1. HTTP. Позволяет взаимодействовать со службами WCF отовсюду, включая Интернет. Коммуникации HTTP можно использовать для создания веб-служб WCF.

2. TCP. Позволяет взаимодействовать со службами WCF в локальной сети или через Интернет, если соответствующим образом сконфигурирован брандмауэр. Протокол TCP является более эффективным, чем HTTP, и обладает большими возможностями, но его несколько сложнее настраивать.
3. Именованный канал. Позволяет взаимодействовать со службами WCF, которые находятся на той же машине, что и вызывающий код, но в отдельном процессе.
4. MSMQ. Это технология очередей, которая позволяет маршрутизировать сообщения, отправленные приложением в место назначения, с помощью очереди. MSMQ — надежная технология обмена сообщениями, которая гарантирует, что сообщение, посланное в очередь, достигнет этой очереди. По природе своей протокол MSMQ также асинхронный, поэтому сообщения из очереди будут обработаны только после обработки предшествующих им сообщений и когда служба-обработчик доступна.

Эти протоколы часто позволяют устанавливать безопасные соединения. Например, протокол HTTPS можно использовать для установки безопасного SSL-соединения через Интернет. Протокол TCP предоставляет расширенные возможности обеспечения безопасности в локальной сети, используя подсистему безопасности Windows.

На рис. 26.1 показано, как с помощью этих транспортных протоколов подключать приложение к службам WCF в различных местоположениях. В этой главе рассматриваются все протоколы кроме MSMQ, который требует более глубокого анализа.

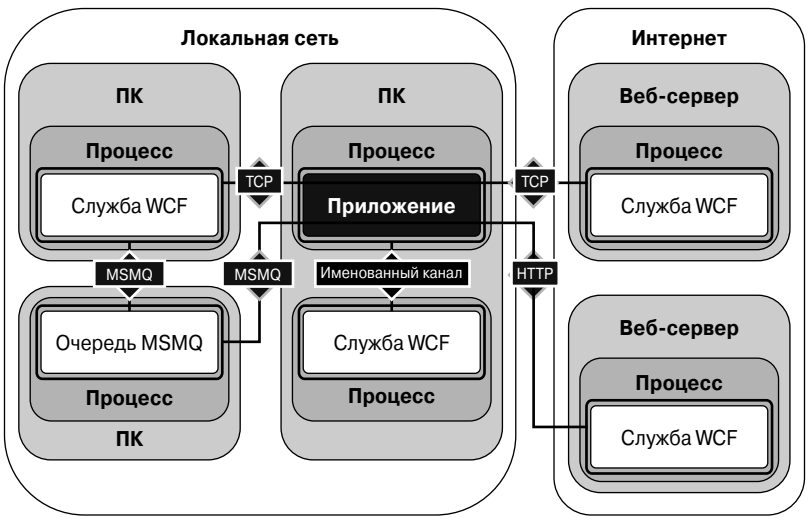


Рис. 26.1. *Подключение приложения к службе WCF*

Для того чтобы подключиться к службе WCF, необходимо знать, где она находится. На практике это означает знание адреса конечной точки.

Адреса, конечные точки и привязки

Тип используемого службой адреса зависит от применяемого протокола. Для трех протоколов, описанных в настоящей главе (MSMQ не рассматривается), адреса служб имеют следующие форматы.

- **HTTP.** Адреса для протокола HTTP — это URL в хорошо знакомой форме `http://<сервер>:<порт>/<служба>`. Для SSL-соединений можно также исполь-

зовать URL вида `https://<сервер>:<порт>/<служба>`. Если вы развертываете службу в IIS, то *<служба>* будет файлом с расширением `.svc` (файлы `.svc` — аналог файлов `.asmx`, используемых веб-службами). Адреса IIS, скорее всего, будут включать множество подкаталогов, т.е. несколько разделов, отделяемых друг от друга символами `/`, перед файлом `.svc`.

- **TCP.** Адреса TCP имеют форму `net.tcp://<сервер>:<порт>/<служба>`.
- **Именованный канал.** Адреса для соединений по именованному каналу похожи, но не включают номер порта. Они имеют форму `net.pipe://<сервер>/<служба>`.

Адрес службы — это базовый адрес, который можно использовать для создания адресов конечных точек соответствующих операций. Например, операция может быть доступна по адресу `net.tcp://<сервер>:<порт>/<служба>/operation1`.

Например, предположим, что была создана служба WCF с единственной операцией, имеющей привязки ко всем трем перечисленным протоколам. В этом случае допускается использовать следующие базовые адреса:

```
http://www.mydomain.com/services/amazingservices/mygreatservice.svc
net.tcp://myhugeserver:8080/mygreatservice
net.pipe://localhost/mygreatservice
```

Для доступа к операции можно применять такие адреса:

```
http://www.mydomain.com/services/amazingservices/mygreatservice.svc/greatop
net.tcp://myhugeserver:8080/mygreatservice/greatop
net.pipe://localhost/mygreatservice/greatop
```

В .NET 4 разрешено использование конечных точек по умолчанию для операций без явного их конфигурирования. Это упрощает настройку, особенно в ситуациях, когда применяются стандартные адреса конечных точек, как в предыдущих примерах.

Привязки, как упоминалось ранее, специфицируют нечто большее, чем просто транспортный протокол, который будет использован операцией. Их можно также применять для указания требований к безопасности, которые должны соблюдаться при коммуникациях через транспортный протокол, транзакционных возможностей конечной точки, кодировки сообщений и многого другого.

Поскольку привязки обеспечивают столь высокую степень гибкости, в .NET Framework предлагается ряд предопределенных привязок, которыми можно пользоваться. Они также пригодны в качестве стартовых значений, которые впоследствии настраиваются для получения необходимого типа привязки — вплоть до точки. Предопределенные привязки обладают заданными возможностями, которые должны быть учтены. Каждый тип привязки представлен классом из пространства имен `System.ServiceModel`. В табл. 26.1 перечислены привязки вместе с некоторой связанной с ними базовой информацией.

Таблица 26.1. Предопределенные привязки WCF

Привязка	Описание
`BasicHttpBinding`	Простейшая привязка HTTP, используемая веб-службами по умолчанию. Обладает ограниченными средствами безопасности и не поддерживает транзакции
`WSHttpBinding`	Более развитая форма привязки HTTP, способная использовать дополнительную функциональность, представленную в WSE
`WSDualHttpBinding`	Расширяет средства `WSHttpBinding` возможностями включения дуплексных коммуникаций. При дуплексной коммуникации сервер может инициировать взаимодействие с клиентом вдобавок к обычному обмену сообщениями

Окончание табл. 26.1

Привязка	Описание
`WSFederationHttpBinding`	Расширяет `WSHttpBinding` федеративными средствами. Федерация позволяет третьим сторонам реализовать средства одиночной регистрации (single sign-on) и другими патентованными мерами безопасности. Это — отдельная тема, которая в настоящей главе не рассматривается
`NetTcpBinding`	Используется для TCP-коммуникаций и позволяет конфигурировать безопасность, транзакции и т.д.
`NetNamedPipeBinding`	Используется для коммуникаций по именованным каналам и позволяет конфигурировать безопасность, транзакции и т.д.
`NetPeerTcpBinding`	Допускает широковещательные коммуникации с множеством клиентов и предоставляет расширенный класс, который в настоящей главе не рассматривается
`NetMsmqBinding` и `MsmqIntegrationBinding`	Эти привязки используются с протоколом MSMQ и в настоящей главе не рассматриваются
`NetPeerTcpBinding`	Используется для одноранговой (peer-to-peer) привязки, которая в настоящей главе не рассматривается
`WebHttpBinding`	Используется для веб-служб, которые применяют запросы HTTP вместо сообщений SOAP
`NetTcpContextBinding`	Подобна `NetTcpBinding`, но позволяет передавать в заголовках SOAP контекстную информацию
`BasicHttpContextBinding` и `WSHttpContextBinding`	Подобны `BasicHttpBinding` и `WSHttpBinding`, но позволяют передавать контекстную информацию соответственно в cookie-наборах HTTP и заголовках SOAP

Многие из перечисленных в этой таблице классов привязки имеют схожие свойства, которые можно использовать для дополнительного конфигурирования. Например, они имеют свойства для установки значения таймаута. При разборе кода примеров далее в главе вы будет приведены более подробные сведения о них.

В .NET 4 конечные точки имеют привязки по умолчанию, которые варьируются в соответствии с применяемым протоколом (табл. 26.2).

Таблица 26.2. Привязки по умолчанию

Протокол	Привязка по умолчанию
HTTP	`BasicHttpBinding`
TCP	`NetTcpBinding`
Именованный канал	`NetNamedPipeBinding`
MSMQ	`NetMsmqBinding`

Контракты

Контракты определяют то, как могут использоваться службы WCF. Доступно несколько типов контрактов.

- **Контракт службы.** Содержит общую информацию о службе и операциях, представленных службой. Включает, например, пространство имен, используемое службой. Служба имеет уникальные пространства имен, которые используются при определении схемы SOAP-сообщений, чтобы избежать возможных конфликтов с другими службами.
- **Контракт операции.** Определяет способ использования операции. Сюда входят типы параметров и возвращаемых значений метода операции наряду с дополнительной информацией, такой как то, возвращает ли метод ответное сообщение.
- **Контракт сообщения.** Позволяет настраивать форматирование информации в сообщениях SOAP, например, должны ли данные включаться в заголовок SOAP либо в тело сообщения SOAP. Это может быть полезно при создании службы WCF, которая должна интегрироваться с унаследованными системами.
- **Контракт сбоя.** Определяет сбои, которые может возвращать операция. При использовании клиентов .NET сбои проявляются в исключениях, которые можно перехватить и обработать нормальным образом.
- **Контракт данных.** Если в качестве типов параметров и типов возврата применяются сложные типы, такие как определяемые пользователем структуры и объекты, то для этих типов должны быть определены контракты данных. Контракты данных определяют типы в терминах данных, представленных в свойствах.

Обычно контракты добавляются к классам служб и их методам с использованием атрибутов, как будет показано далее в главе.

Шаблоны сообщений

В предыдущем разделе вы видели, что контракт операции может определять, возвращает ли операция значение. Вы также уже читали о дуплексных коммуникациях, которые обеспечиваются привязкой `WSDualHttpBinding`. То и другое является шаблоном сообщений (message pattern), которых всего есть три типа.

- **Обмен сообщениями "запрос/ответ".** "Обычный" способ обмена сообщениями, при котором за каждым сообщением, отправленным службе, следует ответное сообщение, отправленное клиенту. Это не обязательно означает, что клиент ждет ответа, поскольку операции могут вызываться и асинхронно.
- **Однонаправленный, или симплексный, обмен сообщениями.** Сообщения отправляются от клиента к операции WCF, но ответы отсутствуют. Это удобно, когда никакие ответы не требуются. Например, можно создать операцию WCF, которая выполняет перезагрузку хост-сервера WCF; в этом случае действительно незачем ждать ответа.
- **Двунаправленный, или дуплексный, обмен сообщениями.** Более развитая схема, при которой клиент в действительности выступает и сервером, и клиентом, а сервер — и клиентом, и сервером. Будучи установленным, дуплексный обмен сообщениями позволяет клиенту и серверу посылать друг другу сообщения, которые могут иметь, а могут и не иметь ответов. Это аналог созданию объекта и подписи на события, предоставляемые этим объектом.

Применение этих шаблонов обмена сообщениями на практике демонстрируется далее в этой главе.

Поведения

Под поведением (behavior) понимается способ применения дополнительной конфигурации, которая не предоставляется непосредственно клиентам службами и операциями.

Добавляя поведение к службе, можно управлять созданием ее экземпляров и использованием ее хостинг-процессами, ее участием в транзакциях, разрешением проблем многопоточности в службе и т.п. С помощью поведений операции можно управлять использованием заимствования прав при выполнении операции, влиянием индивидуальной операции на транзакции и т.д.

В .NET 4 на различных уровнях доступно поведение по умолчанию, так что указывать абсолютно все аспекты поведений каждой службы и операции не понадобится. Вместо этого можно выбрать установки по умолчанию и переопределять их, где требуется, что сократит объем необходимой конфигурации.

Поскольку эта глава призвана предоставить базовые сведения о службах WCF, здесь будет демонстрироваться лишь самая элементарная функциональность поведения.

Хостинг

Во введении в эту главу вы узнали, что службы WCF могут развертываться в нескольких разных процессах. Ниже перечислены имеющиеся возможности.

- **Веб-сервер.** Развернутые в IIS службы WCF ближе всего к веб-службам, предоставляемым WCF. Однако в службах WCF можно использовать расширенную функциональность и средства безопасности, которые гораздо сложнее реализовать в веб-службах. Возможна также интеграция со службами IIS, например, со службой безопасности.
- **Исполняемая программа.** Служба WCF может быть развернута в приложении любого типа, которое допускается создавать в .NET, например, в консольном приложении, приложении Windows Forms и приложении WPF.
- **Служба Windows.** Службу WCF можно развернуть в службе Windows, а это означает возможность использования всех полезных средств, предоставляемых службами Windows. Сюда входит автоматический запуск и восстановление после сбоев.
- **Windows Activation Service (WAS).** Спроектированная специально для развертывания служб WCF, служба WAS (Windows Activation Service — служба активации Windows) — это упрощенная версия IIS, которую можно использовать там, где сервер IIS не доступен.

Две опции из приведенного списка, IIS и WAS, предоставляют полезные средства для служб WCF, такие как активация, повторное использование процессов и пул объектов. Если вы используете любой из других вариантов хостинга, то говорят, что в этих случаях служба WCF обладает *собственным хостингом*. Это не обязательно плохо, поскольку дополнительная функциональность, которую предлагает развитая хостинг-среда, может быть просто не нужна. Однако службы с собственным хостингом требуют написания большего объема кода.

Программирование с использованием WCF

Ознакомившись с основами, теперь самое время приступить к работе с некоторым кодом. В этом разделе мы начнем с рассмотрения простой службы WCF, развернутой на веб-сервере, и клиентского консольного приложения. После просмотра структуры созданного кода будет представлена базовая структура служб WCF и клиентских приложений. Затем мы более подробно рассмотрим некоторые ключевые темы, в том числе:

- определение контрактов служб WCF;
- службы WCF с собственным хостингом.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Простая служба и клиент WCF

1. Создайте новый проект WCF Service Application (Приложение службы WCF) по имени `Ch26Ex01` и сохраните его в каталоге `C:\BegVCSharp\Chapter26`.
2. Добавьте к решению консольное приложение по имени `Ch26Ex01Client`.
3. Выберите пункт меню Build⇨Build Solution (Компоновка⇨Собрать решение).
4. Щелкните правой кнопкой мыши на проекте `Ch26Ex01Client` в окне Solution Explorer и выберите в контекстном меню пункт Add Service Reference (Добавить ссылку на службу).
5. В диалоговом окне Add Service Reference (Добавление ссылки на службу) щелкните на Discover (Обнаружить).
6. После запуска веб-сервера разработки и загрузки информации о службе WCF, раскройте ссылку, чтобы увидеть детали, как показано на рис. 26.2 (номер порта может отличаться).

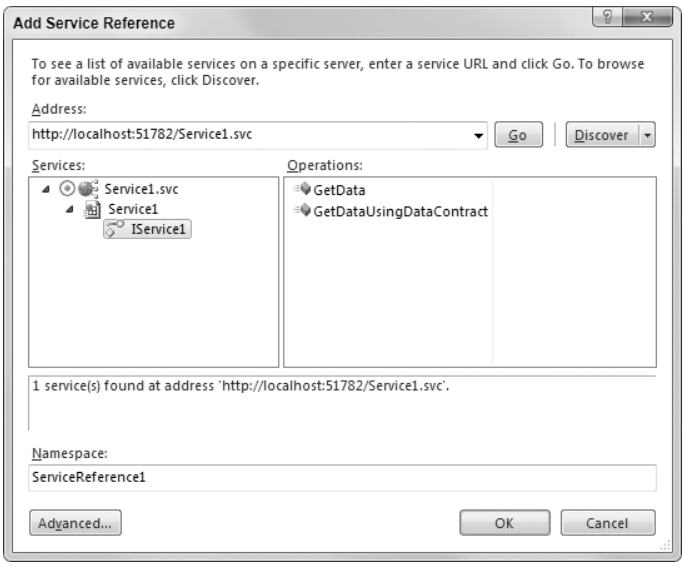


Рис. 26.2. *Диалоговое окно* Add Service Reference

7. Щелкните на кнопке OK для добавления ссылки на службу.
8. Модифицируйте код в `Program.cs` в приложении `Ch26Ex01Client`, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch26Ex01Client.ServiceReference1;
namespace Ch26Ex01Client
{
  class Program
  {
```

```
    static void Main(string[] args)
    {
      string numericInput = null;
      int intParam;
      do
      {
        Console.WriteLine(
            "Enter an integer and press enter to call the WCF service.");
        // Введите целое число и нажмите Enter для обращения к службе WCF
        numericInput = Console.ReadLine();
      }
      while (!int.TryParse(numericInput, out intParam));
      Service1Client client = new Service1Client();
      Console.WriteLine(client.GetData(intParam));
      Console.WriteLine("Press an key to exit.");
      Console.ReadKey();
    }
  }
}
```

Фрагмент кода ***Ch26Ex01Client\Program.cs***

9. Щелкните правой кнопкой мыши на проекте `Ch2601Client` в окне Solution Explorer и выберите в контекстном меню пункт Set as StartUp Project (Установить как стартовый проект).
10. Запустите приложение. В ответ на приглашение щелкните на кнопке OK, чтобы включить отладку в `Web.config`. Введите целое число в окне консольного приложения и нажмите <Enter>. Результат показан на рис. 26.3.

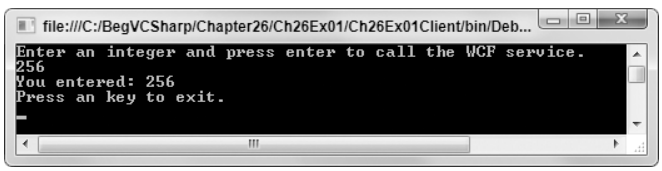


Рис. 26.3. *Результат обращения к службе WCF из консольного приложения*

11. Выйдите из приложения, щелкните правой кнопкой мыши на файле `Service1.svc` проекта `Ch26Ex01` в окне Solution Explorer и выберите в контекстном меню пункт View in Browser (Просмотреть в браузере).
12. Просмотрите информацию в окне браузера (рис. 26.4).
13. Щелкните на ссылке в верхней части веб-страницы службы, чтобы просмотреть WSDL-описание. Знать, что означает код в файле WSDL, вовсе не обязательно.

Описание работы

В этом примере была создана простая служба WCF, развернутая на веб-сервере, и клиентское консольное приложение. Мы использовали шаблон VS по умолчанию для проекта службы WCF, а это значит, что добавлять какой-либо код не понадобится. Вместо этого использовалась одна из операций, определенных в шаблоне по умолчанию — `GetData()`. Для целей настоящего примера действительно применяемая операция не важна; здесь внимание сосредоточено на структуре кода и механизмах, которые обеспечивают всю работу.

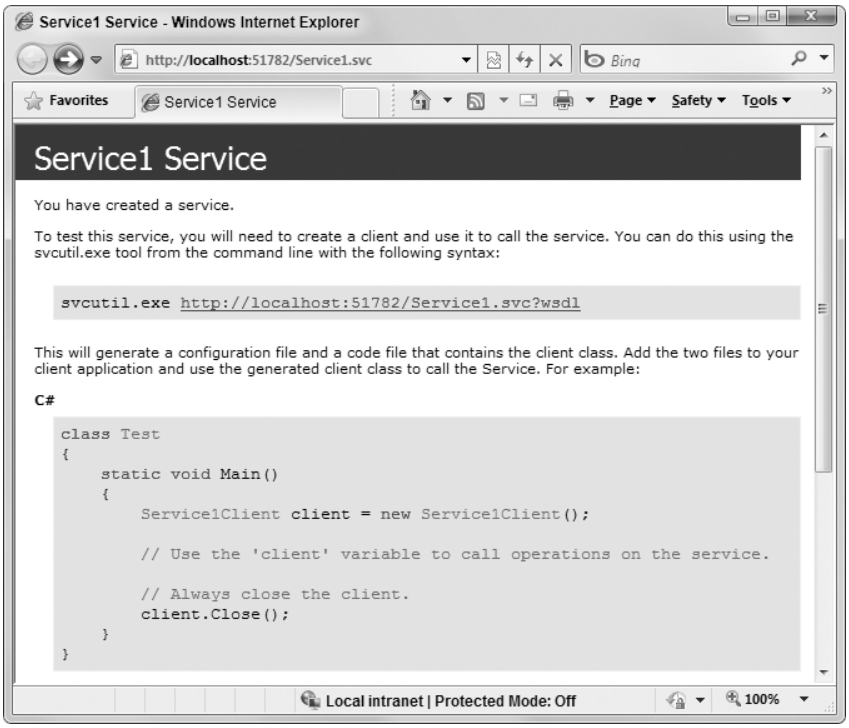


Рис. 26.4. *Просмотр информации о службе*

Для начала рассмотрим проект сервера — `Ch26Ex01`. Он состоит из следующих компонентов.

- Файл `Service1.svc`, определяющий хостинг для службы.
- Определение класса `CompositeType`, который представляет контракт данных, используемый службой (в файле кода `IService1.cs`).
- Определение интерфейса `IService1`, который задает контракт службы и два контракта операций для службы.
- Определение класса `Service1`, который реализует `IService1` и определяет функциональность службы (в файле кода `Service1.svc`).
- Раздел конфигурации службы `<system.serviceModel>` (в файле `Web.config`).

Файл `Service1.svc` содержит следующую строку кода (чтобы просмотреть код, щелкните правой кнопкой мыши на файле в окне Solution Explorer и выберите в контекстном меню пункт View Markup (Просмотреть разметку)):

```
<%@ ServiceHost Language="C#" Debug="true" Service="Ch26Ex01.Service1"
    CodeBehind="Service1.svc.cs" %>
```

Фрагмент кода ***Ch26Ex01\Service1.svc***

Это инструкция `ServiceHost`, используемая для сообщения веб-серверу (веб-серверу разработки в данном случае, хотя это также применимо к IIS), какая служба развернута по этому адресу. Класс, определяющий службу, объявлен в атрибуте `Service`, а файл кода, определяющий этот класс, — в атрибуте `CodeBehind`. Эта инструкция необходима для того, чтобы получить средства хостинга веб-службы, как указано в предшествующих разделах.

Очевидно, что этот файл не нужен для служб WCF, которые не развернуты на веб-сервере. О развертывании служб WCF с собственным хостингом вы узнаете далее в этой главе.

После этого в файле IService1.cs определяется контракт данных CompositeType. Вы коде видно, что контракт данных — это просто определение класса, которое содержит атрибут DataContract в определении класса и атрибуты DataMember — в его членах:

```
[DataContract]
public class CompositeType
{
  bool boolValue = true;
  string stringValue = "Hello ";

  [DataMember]
  public bool BoolValue
  {
    get { return boolValue; }
    set { boolValue = value; }
  }

  [DataMember]
  public string StringValue
  {
    get { return stringValue; }
    set { stringValue = value; }
  }
}
```

Фрагмент кода ***Ch26Ex01\IService1.cs***

Этот контракт данных представлен клиентскому приложению через метаданные (если вы заглядывали в файл WSDL примера, то могли видеть это). Это позволяет клиентским приложениям определять тип, который может быть сериализован в форму, которая может быть десериализована службой в объект CompositeType. Клиенту не обязательно знать реальное определение типа; фактически класс, используемый клиентом, может иметь другую реализацию. Этот простой способ определения контрактов данных неожиданно мощный и позволяет организовать передачу сложных структур данных между службой WCF и ее клиентами.

Файл IService1.cs также содержит контракт службы, определенный как интерфейс с атрибутом ServiceContract. Опять-таки, этот интерфейс полностью описан в метаданных службы и может быть воссоздан в клиентском приложении. Члены интерфейса состоят из операций, предоставляемых службой, каждая из которых используется для создания контракта операции применением атрибута OperationContract. Пример кода содержит две операции, одна из которых использует контракт данных, встречавшийся ранее:

```
[ServiceContract]
public interface IService1
{
  [OperationContract]
  string GetData(int value);

  [OperationContract]
  CompositeType GetDataUsingDataContract(CompositeType composite);
}
```

Все четыре определяющих контракт атрибута, которые мы видели до сих пор, могут быть дополнительно сконфигурированы атрибутами, как показано в следующем разделе. Код, реализующий службу, выглядит подобно любому другому определению класса:

```
public class Service1 : IService1
{
  public string GetData(int value)
  {
    return string.Format("You entered: {0}", value);
  }
  public CompositeType GetDataUsingDataContract(CompositeType composite)
  {
    if (composite == null)
    {
      throw new ArgumentNullException("composite");
    }
    if (composite.BoolValue)
    {
      composite.StringValue += "Suffix";
    }
    return composite;
  }
}
```

Фрагмент кода ***Ch26Ex01\Service1.svc.cs***

Обратите внимание, что это определение класса не обязано наследоваться от какого-то определенного типа, и не требует никаких специальных атрибутов. Все, что необходимо сделать — это реализовать интерфейс, определяющий контракт службы. Фактически к этому классу и его членам можно добавлять атрибуты, чтобы указать поведения, хотя это не обязательно.

Отделение контракта службы (интерфейса) от ее реализации (класса) работает исключительно хорошо. Клиенту не надо знать ничего о классе, который может включать намного больше функциональности, чем простая реализация службы. Один класс может даже реализовывать более одного контракта службы.

Наконец, обратимся к конфигурации в файле `Web.config`. Настройка служб WCF в конфигурационных файлах позаимствована в .NET Remoting, и она работает со всеми типами служб WCF, а также с клиентами служб WCF (как будет показано ниже). Словарь этой конфигурации настолько богат, что позволяет настроить все, что понадобится настраивать в службе, и его синтаксис даже можно расширять сверх этого.

Код конфигурации WCF содержится в разделе `<system.serviceModel>` файлов `Web.config` и `app.config`. В данном примере конфигурирования службы не так уж много, поскольку используются значения по умолчанию. В файле `Web.config` раздел конфигурации состоит из единственного подраздела, который представляет переопределения значений по умолчанию для поведения службы `<behaviors>`. Код раздела конфигурации `<system.serviceModel>` в файле `Web.config` (комментарии удалены для ясности) выглядит так:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceMetadata httpGetEnabled="true"/>
        <serviceDebug includeExceptionDetailInFaults="false"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

Фрагмент кода ***Ch26Ex01\Web.config***

В этом разделе может быть определено одно или больше поведений в дочернем разделе <behavior>, которые могут быть повторно использованы во множестве других элементов. Разделу <behavior> можно назначить имя для облегчения повторного использования (чтобы на него можно было ссылаться отовсюду), или же он может быть использован без имени (как в этом примере), чтобы указать переопределения установок поведения по умолчанию.

Если бы применялась конфигурация не по умолчанию, можно было бы ожидать наличия внутри <system.serviceModel> раздела <services>, содержащего один или более дочерних разделов <service>. В свою очередь, раздел <service> может содержать дочерние разделы <endpoint>, каждый из которых определяет конечную точку для службы. Из них выводятся конечные точки для операций.

Ниже показано одно из определений поведения по умолчанию в Web.config:

```
<serviceDebug includeExceptionDetailInFaults="false" />
```

Эта настройка может быть установлена в true, чтобы предоставить детали исключения при любом сбое, который передается клиенту — нечто такое, что обычно возможно только при разработке.

Другое поведение по умолчанию, которое переопределяется в Web.config, связано с метаданными. Метаданные используются для того, чтобы позволить клиентам получать описания служб WCF. В конфигурации по умолчанию определены две конечные точки служб по умолчанию. Одна — конечная точка, которую клиенты применяют для доступа к службе, а другая — конечная точка, служащая для получения метаданных от службы. Это можно отключить в файле Web.config, как показано ниже:

```
<serviceMetadata httpGetEnabled="false" />
```

В качестве альтернативы можно было бы полностью удалить эту строку из конфигурационного файла, поскольку поведение по умолчанию не разрешает передачу метаданных.

Если вы попробуете такое отключение в рассматриваемом примере, это не прекратит доступ клиента к службе, потому что он уже получил необходимые ему метаданные при добавлении ссылки на службу. Однако отключение метаданных помешает другим клиентам использовать инструмент Add Service Reference (Добавление ссылки на службу) для данной службы. Обычно веб-службы в рабочих средах не нуждаются в предоставлении метаданных, так что по завершении разработки эту функциональность можно спокойно отключить.

Кроме метаданных есть и другой распространенный способ доступа к службе WCF — определение контракта в отдельной сборке, на которую ссылается как хостинг-проект, так и проект клиента. Клиент может тогда вместо предоставления метаданных сгенерировать прокси, используя эти контракты непосредственно.

Теперь, когда мы рассмотрели код службы WCF, настало время взглянуть на клиента, в частности, на то, что действительно делает инструмент Add Service Reference. В окне Solution Explorer вы заметите, что клиент включает папку по имени Service References, раскрыв которую, вы увидите элемент по имени ServiceReference1 — имя, которое было выбрано при добавлении ссылки.

Инструмент Add Service Reference создает все классы, которые понадобятся для доступа к службе. Сюда входит класс прокси для службы, которая включает методы для всех операций, предоставляемых службой (Service1Client), и класс клиентской стороны, сгенерированный из контракта данных (CompositeType).

При желании можно просмотреть код, сгенерированный инструментом Add Service Reference (отобразив все файлы проекта, включая скрытые), хотя сейчас, вероятно, не стоит этого делать, потому что они содержат массу сложного кода.

Инструмент также добавляет в проект конфигурационный файл `app.config`. Эта конфигурация определяет два аспекта:

- информация привязки конечной точки службы;
- адрес и контракт для конечной точки.

Информация привязки берется из описания службы, и на клиенте каждая отдельная конфигурируемая опция копируется в конфигурационный файл:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_IService1"
          closeTimeout="00:01:00" openTimeout="00:01:00"
          receiveTimeout="00:10:00" sendTimeout="00:01:00"
          allowCookies="false" bypassProxyOnLocal="false"
          hostNameComparisonMode="StrongWildcard"
          maxBufferSize="65536" maxBufferPoolSize="524288"
          maxReceivedMessageSize="65536"
          messageEncoding="Text" textEncoding="utf-8"
          transferMode="Buffered" useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
            maxArrayLength="16384" maxBytesPerRead="4096"
            maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None"
              proxyCredentialType="None" realm="" />
            <message clientCredentialType="UserName"
              algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
```

Фрагмент кода ***Ch26Ex01Client\app.config***

Эта привязка используется в конфигурации конечной точки, наряду с базовым адресом службы (которым является адрес файла `.svc` для служб, развернутых на веб-сервере) и версией контракта `IService1` клиентской стороны:

```
    <client>
      <endpoint address="http://localhost:51782/Service1.svc"
        binding="basicHttpBinding"
        bindingConfiguration="BasicHttpBinding_IService1"
        contract="ServiceReference1.IService1"
        name="BasicHttpBinding_IService1" />
    </client>
  </system.serviceModel>
</configuration>
```

Инструмент Add Service Reference здесь чрезмерно постарался. Фактически большая часть этой информации не нужна. Содержимое этого конфигурационного файла можно заменить показанным ниже:

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address="http://localhost:51782/Service1.svc"
        binding="basicHttpBinding"
        contract="ServiceReference1.IService1"
        name="BasicHttpBinding_IService1" />
```

```
    </client>
  </system.serviceModel>
</configuration>
```

Весь раздел <binding> вместе с атрибутом bindingConfiguration элемента <endpoint> исключен, а это означает, что клиент использует конфигурацию привязки по умолчанию.

Однако для изучения служб WCF такое старание инструмента очень кстати. Он демонстрирует все установки, которые включает привязка BasicHttpBinding по умолчанию. В этой главе мы не слишком углубляемся в детали конфигурации службы WCF, но вы уже можете видеть некоторые из них — например, настройки таймаута, которые легко понять благодаря их явному наименованию (их имена содержат слово "Timeout").

Рассмотренный пример охватывает основы, и прежде чем двигаться дальше, стоит подытожить изученное:

- Определения службы WCF
  - Службы определяются интерфейсом контракта службы, включающим операции — члены контракта.
  - Службы реализованы в классе, реализующем интерфейс контракта службы
  - Контракты данных — это просто определения типов, использующих атрибуты контракта данных
- Конфигурация службы WCF
  - Для конфигурирования служб WCF можно применять конфигурационные файлы (Web.config или app.config)
- Хостинг WCF на веб-сервере
  - При хостинге на веб-сервере в качестве базового адреса службы используются файлы .svc
- Конфигурация клиента WCF
  - Для конфигурирования клиентов служб WCF можно применять конфигурационные файлы (Web.config или app.config)

В следующем разделе контракты рассматриваются более подробно.

Тестовый клиент WCF

В предыдущем практическом занятии были созданы служба и клиент, что позволило увидеть, как работает базовая архитектура WCF и как осуществляется конфигурирование служб WCF. На практике, однако, используемое клиентское приложение может быть сложным, и правильно протестировать службы оказывается непросто.

Для облегчения разработки служб WCF в VS предлагается инструмент тестирования, который можно применять для проверки корректности работы операций, предоставляемых службой WCF. Этот инструмент автоматически сконфигурирован для работы с проектами служб WCF, так что он открывается после запуска проекта. Все, что потребуется сделать — это удостовериться, что служба, которую необходимо тестировать (т.е. файл .svc) установлена как стартовая страница для проекта службы WCF. В качестве альтернативы тестовый клиент можно запустить как автономное приложение. Тестовый клиент для 64-битных операционных систем находится в каталоге C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\WcfTestClient.exe.

В случае 32-битной операционной системы путь будет таким же, за исключением корневой папки — Program Files. Данный инструмент позволяет вызывать операции службы и просматривать службы рядом других способов. Это иллюстрируется в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование тестового клиента WCF

1. Откройте проект WPF Service Application из предыдущего практического занятия — `Ch26Ex01`.
2. Щелкните правой кнопкой мыши на службе `Service1.svc` в окне Solution Explorer и выберите в контекстном меню пункт Set As Start Page (Установить как стартовую страницу).
3. Щелкните правой кнопкой мыши на проекте `Ch26Ex01` в окне Solution Explorer и выберите в контекстном меню пункт Set As StartUp Project (Установить как стартовый проект).
4. В `Web.config` включите метаданные:

```
<serviceMetadata httpGetEnabled="true" />
```

Фрагмент кода ***Ch26Ex01\Web.config***

5. Запустите приложение. Появится тестовый клиент WCF, показанный на рис. 26.5 (пара секунд уйдет на добавление службы).

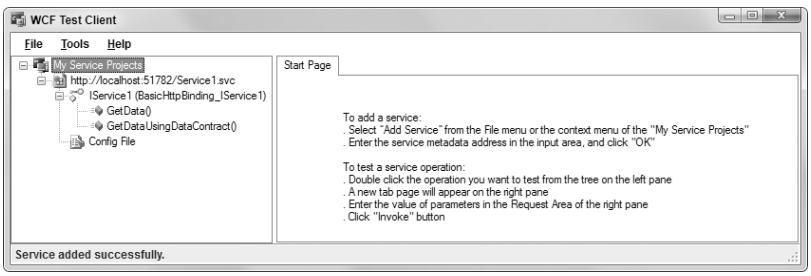


Рис. 26.5. *Тестовый клиент WCF*

6. В левой панели тестового клиента дважды щелкните на элементе Config File (Файл конфигурации). При этом в правой панели отобразится содержимое конфигурационного файла, используемого для доступа к службе (рис. 26.6).

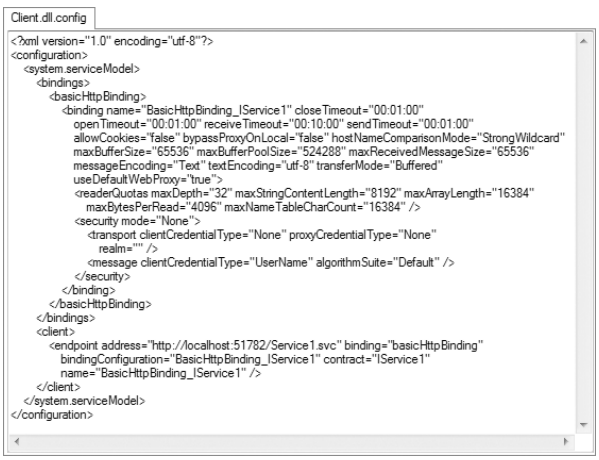


Рис. 26.6. *Отображение содержимого конфигурационного файла*

7. В левой панели дважды щелкните на операции `GetDataUsingDataCOntract()`.
8. В панели, которая появится справа, измените значение `BoolValue` на `True`, а `StringValue` — на `Test String`, и щелкните на кнопке Invoke (Вызвать).
9. Если откроется диалоговое окно с предупреждением о безопасности, щелкните на кнопке OK для подтверждения отправки информации службе.
10. Отобразится результат операции, как показано на рис. 26.7.

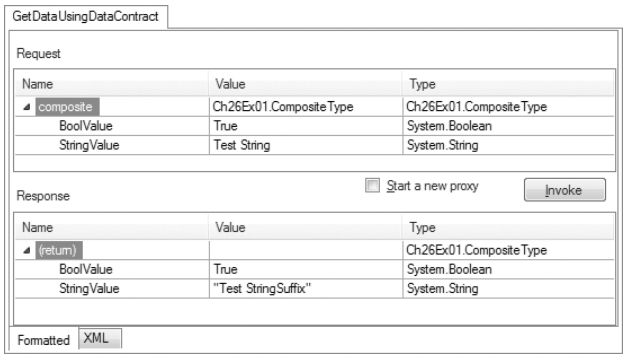


Рис. 26.7. *Отображение результата операции*

11. Щелкните на вкладке XML для отображения XML-кода запроса и ответа (рис. 26.8).

Рис. 26.8. *Отображение XML-кода запроса и ответа*

Описание работы

В этом примере тестовый клиент WCF использовался для просмотра и вызова операции из службы, созданной в предыдущем практическом занятии. Первое, что вы, скорее всего, заметите — это небольшая задержка при загрузке службы. Она связана с тем, что тестовый клиент должен исследовать службу, чтобы определить ее возможности. Это исследование использует те же метаданные, что и инструмент Add Service Reference — именно поэтому должен быть включен доступ к метаданным (возможно, он был отключен во время экс-

периментов в предыдущем практическом занятии). Как только обнаружение завершено, служба и ее операции появляются в левой панели инструмента.

Затем вызывается операция. Тестовый клиент позволяет вводить параметры для использования и вызова метода с последующим отображением результата — и все это без написания клиентского кода. Также было показано, как просматривать действительный XML, который был отправлен и получен в виде результата. Эта информация носит технический характер, однако она может быть чрезвычайно важна при отладке более сложных служб.

Определение контрактов службы WCF

Предыдущие примеры продемонстрировали, как инфраструктура WCF облегчает определение контрактов для служб WCF в виде комбинации классов, интерфейсов и атрибутов. В этом разделе мы рассмотрим это более внимательно.

Контракты данных

Чтобы определить контракт данных для службы, к определению класса применяется атрибут `DataCOntractAttribute`. Этот атрибут находится в пространстве имен `System.Runtime.Serialization`. Его можно конфигурировать с помощью свойств, описанных в табл. 26.3.

Таблица 26.3. Свойства атрибута `DataContractAttribute`

Свойство	Описание
`Name`	Назначает контракту данных имя, отличающееся от имени, которое было указано в определении класса. Это имя используется в сообщениях SOAP и объектах данных клиентской стороны, определенных по метаданным службы
`Namespace`	Определяет пространство имен, используемое контрактом данных в сообщениях SOAP

Оба эти свойства полезны, когда нужна совместимость с существующими форматами сообщений SOAP (как и аналогично именованные свойства для других контрактов), но в противном случае они, скорее всего, не понадобятся.

Каждый член класса, являющийся частью контракта данных, должен использовать атрибут `DataMemberAttribute`, который также находится в пространстве имен `System.Runtime.Serialization`. Этот атрибут имеет свойства, описанные в табл. 26.4.

Таблица 26.4. Свойства атрибута `DataMemberAttribute`

Свойство	Описание
`Name`	Указывает имя члена данных при сериализации (по умолчанию — имя члена)
`IsRequired`	Указывает, должен ли член присутствовать в сообщении SOAP
`Order`	Значение `int`, указывающее порядок сериализации или десериализации члена, который может понадобиться, если один член должен быть представлен перед другим. Члены с более низкими значениями `Order` обрабатываются первыми
`Namespace`	Установите это свойство в `false`, чтобы предотвратить включение членов в сообщения SOAP, если их значения является значениями по умолчанию для каждого данного члена

Контракты службы

Контракты службы определяются с применением атрибута `System.ServiceModel.ServiceContractAttribute` в определении интерфейса. Для настройки контракта службы доступны свойства, перечисленные в табл. 26.5.

Таблица 26.5. Свойства атрибута `System.ServiceModel.ServiceContractAttribute`

Свойство	Описание
`Name`	Указывает имя контракта службы, как она определена в элементе `<portType>` в описании WSDL
`Namespace`	Определяет пространство имен контракта службы, использованное элементом `<portType>` в описании WSDL
`ConfigurationName`	Имя контракта службы, как указано в конфигурационном файле
`HasProtectionLevel`	Определяет, имели ли сообщения, используемые службой, явно определенные уровни защиты. Уровни защиты позволяют подписывать либо подписывать и шифровать сообщения
`ProtectionLevel`	Уровень защиты сообщений
`SessionMode`	Определяет, включены ли сеансы для сообщений. Если вы используете сеансы, то можете гарантировать, что сообщения, посланные в различные конечные точки службы, будут связанными, т.е. использовать один и тот же экземпляр службы, а потому разделять общее состояние, и т.п.
`CallbackContract`	Для дуплексного обмена сообщениями клиент представляет контракт наряду со службой. Это нужно потому, что, как было сказано ранее, в дуплексных коммуникациях клиент также выступает в роли сервера. Это свойство позволяет указывать контракт, который использует клиент

Контракты операций

Внутри интерфейсов, определяющих контракты службы, определяются члены в виде операций за счет применения атрибута `System.ServiceModel.OperationContractAttribute`. Этот атрибут имеет поддерживает свойства, описанные в табл. 26.6.

Таблица 26.6. Свойства атрибута `System.ServiceModel.OperationContractAttribute`

Свойство	Описание
`Name`	Указывает имя операции службы. По умолчанию — это имя члена
`IsOneWay`	Указывает, возвращает ли операция ответ. Если установить это свойство в `true`, то клиенты не будут ожидать завершения операции перед тем, как продолжить работу
`AsyncPattern`	При установке этого свойства в `true` операция реализуется в виде двух методов, которые можно использовать для асинхронного вызова операции: `Begin<`*`имяМетода`*`>()` и `End<`*`имяМетода`*`>()`
`HasProtectionLevel`	См. табл. 26.5
`ProtectionLevel`	См. табл. 26.5

Окончание табл. 26.6

Свойство	Описание
`IsInitiating`	Если сеансы используются, то это свойство определяет, может ли вызов этой операции запустить новый сеанс
`IsTerminating`	Если сеансы используются, то это свойство определяет, может ли вызов этой операции завершить текущий сеанс
`Action`	Если вы используете адресацию (расширенная возможность служб WCF), то операция имеет ассоциированное с ней имя действия, которое можно указать в этом свойстве
`ReplyAction`	Похоже на предыдущее, но задает имя действия для ответа операции

Контракты сообщения

В предыдущем примере спецификации контракта сообщения не применялись. Для их использования должен быть определен класс, представляющий сообщение, и к этому классу должно быть применен атрибут `MessageContractAttribute`. Затем к членам этого класса применяются атрибуты `MessageBodyMemberAttribute`, `MessageHeaderAttribute` или `MessageHeaderArrayAttribute`. Все эти атрибуты находятся в пространстве имен `System.SErviceModel`. Маловероятно, что вам это понадобится, если только не нужна высокая степень контроля над сообщениями SOAP, которые используются службами WCF, поэтому мы не станем здесь углубляться в детали.

Контракты сбоев

При наличии определенного типа исключения, например, специального исключения, которое должно быть сделано доступным клиентским приложениям, можно применить атрибут `System.ServiceModel.FaultContractAttribute` к операции, которая может генерировать это исключение. Опять-таки, это не то, что будет делаться при обычном использовании WCF.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Контракты WCF

1. Создайте новый проект WPF Service Application по имени `Ch26Ex02` в каталоге `C:\BeginVCSharp\Chapter26`.
2. Добавьте к решению проект библиотеки классов по имени `Ch26Ex02Contracts` и удалите файл `Class1.cs`.
3. Добавьте в проект `Ch26Ex02Contracts` ссылки на сборки `System.Runtime.Serialization.dll` и `System.ServiceModel.dll`.
4. Добавьте в проект `Ch26Ex02Contracts` класс по имени `Person` и модифицируйте код в `Person.cs` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
namespace Ch26Ex02Contracts
{
  [DataContract]
```

```
  public class Person
  {
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public int Mark { get; set; }
  }
}
```

Фрагмент кода ***Ch26Ex02Contracts\Person.cs***

5. Добавьте класс по имени `IAwardService` в проект `Ch26Ex02Contracts` и модифицируйте код в `IAwardService.cs`, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
namespace Ch26Ex02Contracts
{
  [ServiceContract(SessionMode=SessionMode.Required)]
  public interface IAwardService
  {
    [OperationContract(IsOneWay=true,IsInitiating=true)]
    void SetPassMark(int passMark);
    [OperationContract]
    Person[] GetAwardedPeople(Person[] peopleToTest);
  }
}
```

Фрагмент кода ***Ch26Ex02Contracts\IAwardService.cs***

6. Добавьте в проект `Ch26Ex02` ссылку на проект `Ch26Ex02Contracts`.
7. Удалите `IService1.cs` и `Service1.svc` из проекта `Ch26Ex02`.
8. Добавьте в проект `Ch26Ex02` новую службу WCF по имени `AwardService.cs`.
9. Удалите файл `IAwardService.cs` из проекта `Ch26Ex02`.
10. Модифицируйте код в `AwardService.svc.cs` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
using Ch26Ex02Contracts;
namespace Ch26Ex02
{
  public class AwardService : IAwardService
  {
    private int passMark;
    public void SetPassMark(int passMark)
    {
      this.passMark = passMark;
    }
    public Person[] GetAwardedPeople(Person[] peopleToTest)
    {
      List<Person> result = new List<Person>();
```

```
        foreach (Person person in peopleToTest)
        {
          if (person.Mark > passMark)
          {
            result.Add(person);
          }
        }
        return result.ToArray();
      }
    }
}
```

Фрагмент кода ***Ch26Ex02\AwardService.svc.cs***

11. Модифицируйте конфигурационный раздел службы в `Web.config`, как показано ниже:

```
<system.serviceModel>
  <protocolMapping>
    <add scheme="http" binding="wsHttpBinding"/>
  </protocolMapping>
  ...
</system.serviceModel>
```

12. Откройте свойства проекта `Ch26Ex02`. В разделе Web выберите переключатель Specific port (Специальный порт) и введите номер порта `51425`, как показано на рис. 26.9.

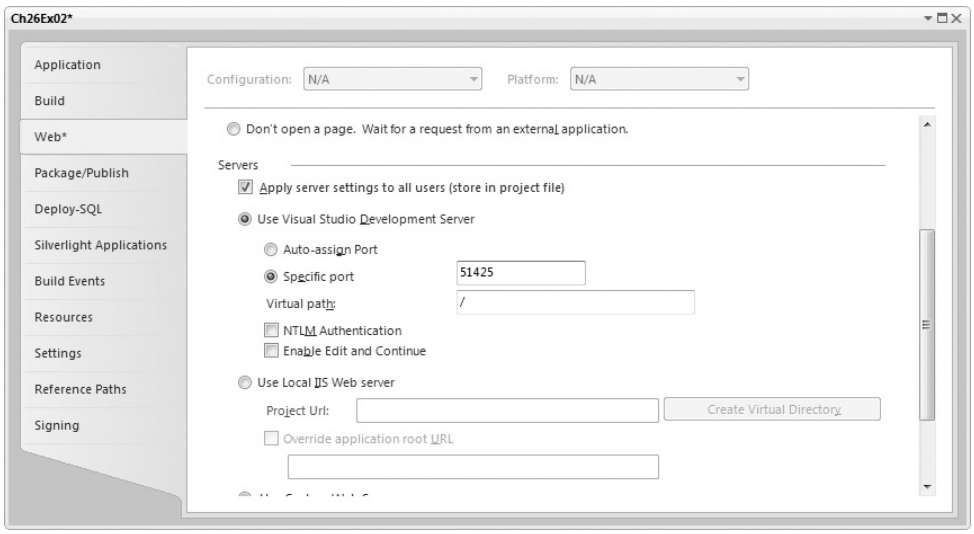


Рис. 26.9. *Указание специального порта*

13. Добавьте к решению новый консольный проект по имени `Ch26Ex02Client` и установите его в качестве стартового проекта.
14. Добавьте в проект `Ch26Ex02Client` ссылки на сборку `System.ServiceModel.dll` и проект `Ch26Ex02Contracts`.
15. Модифицируйте код в файле `Program.cs` в проекте `Ch26Ex02Client` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using Ch26Ex02Contracts;
namespace Ch26E02Client
{
  class Program
  {
    static void Main(string[] args)
    {
      Person[] people = new Person[]
      {
        new Person { Mark = 46, Name="Jim" },
        new Person { Mark = 73, Name="Mike" },
        new Person { Mark = 92, Name="Stefan" },
        new Person { Mark = 84, Name="George" },
        new Person { Mark = 24, Name="Arthur" },
        new Person { Mark = 58, Name="Nigel" }
      };
      Console.WriteLine("People:");
      OutputPeople(people);
      IAwardService client = ChannelFactory<IAwardService>.CreateChannel(
        new WSHttpBinding(),
        new EndpointAddress("http://localhost:51425/AwardService.svc"));
      client.SetPassMark(70);
      Person[] awardedPeople = client.GetAwardedPeople(people);
      Console.WriteLine();
      Console.WriteLine("Awarded people:");
      OutputPeople(awardedPeople);
      Console.ReadKey();
    }
    static void OutputPeople(Person[] people)
    {
      foreach (Person person in people)
      {
        Console.WriteLine("{0}, mark: {1}", person.Name, person.Mark);
      }
    }
  }
}
```

Фрагмент кода ***Ch26Ex02Client\Program.cs***

16. Запустите приложение. Результат показан на рис. 26.10.

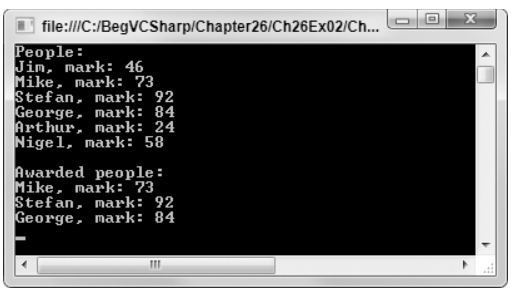


Рис. 26.10*. Пример использования контрактов*

Описание работы

В этом примере был создан набор контрактов в проекте библиотеки классов — как в службе, так и в клиенте. Служба, как в предыдущем примере, развернута на веб-сервере. Конфигурация для этой службы сведена к абсолютному минимуму.

Главное отличие текущего примера в том, что клиенту не требуется никаких метаданных, поскольку клиент имеет доступ к сборке контракта. Вместо генерации прокси-класса из метаданных клиент получает ссылку на интерфейс контракта службы через альтернативный метод. Другой момент, который следует отметить в этом примере, касается использования сеанса для поддержки состояния службы, что потребует привязки `WSHttpBinding` вместо `BasicHttpBinding`.

Контрактом данных в этом примере был простой класс по имени `Person`, у которого есть свойство типа `string` по имени `Name` и свойство типа `int` по имени `Mark`. Атрибуты `DataContractAttribute` и `DataMemberAttribute` применялись без настройки, потому нет нужды здесь еще раз разбирать код этого контракта.

Контракт службы был определен за счет применения атрибута `ServiceContractAttribute` к интерфейсу `IAwardService`. Свойство `SessionMode` этого атрибута было установлено в `SessionMode.Required`, поскольку эта служба требует состояния:

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IAwardService
{
```

Контракт первой операции, `SetPassMark()`, устанавливает состояние, поэтому свойство `IsInitiating` атрибута `OperationContractAttribuite` установлено в `true`. Эта операция не возвращает ничего, поэтому она определена как однонаправленная операция путем установки `IsOneWay` в `true`:

```
  [OperationContract(IsOneWay=true,IsInitiating=true)]
  void SetPassMark(int passMark);
```

Контракт другой операции, `GetAwardedPeople()`, не требует никакой настройки и использует контракт данных, определенный ранее:

```
  [OperationContract]
  Person[] GetAwardedPeople(Person[] peopleToTest);
}
```

Помните, что эти два типа — `Person` и `IAwardService` — доступны как службе, так и клиенту. Служба реализует контракт `IAwardService` в типе по имени `AwardService`, который не содержит никакого примечательного кода. Единственное отличие этого класса от класса службы, который мы видели ранее, состоит в том, что он обладает состоянием. Это допустимо, поскольку сеанс определен для связывания сообщений от клиента.

Чтобы обеспечить использование службой привязки `WSHttpBinding`, для службы был добавлен следующий файл `Web.config`:

```
<protocolMapping>
  <add scheme="http" binding="wsHttpBinding"/>
</protocolMapping>
```

Это переопределяет отображение по умолчанию для привязки HTTP. В качестве альтернативы можно было бы сконфигурировать службу вручную и сохранить существующее умолчание, но такое переопределение намного проще. Однако имейте в виду, что переопределение такого рода применяется ко всем службам проекта. При наличии в проекте более одной службы следует удостовериться, что эта привязка приемлема для каждой из них.

Клиент более интересен — прежде всего, из-за следующей строки кода:

```
IAwardService client = ChannelFactory<IAwardService>.CreateChannel(
  new WSHttpBinding(),
  new EndpointAddress("http://localhost:51425/AwardService.svc"));
```

Клиентское приложение не имеет файла `app.config` для конфигурирования коммуникаций со службой, и для взаимодействия со службой никакого прокси-класса из метаданных не определяется. Вместо этого прокси-класс создается вызовом метода `ChannelFactory<T>.CreateChannel()`. Этот метод создает прокси-класс, реализующий клиент `IAwareService`, хотя "за кулисами" сгенерированный класс взаимодействует со службой точно так же, как сгенерированный из метаданных прокси, показанный ранее.

Если вы создаете прокси-класс с помощью `ChannelFactory<T>.CreateChannel()`, то коммуникационный канал по умолчанию закрывается по таймауту через минуту, что приведет к ошибкам коммуникации. Существуют способы сохранить соединения активными, но их описание выходит за рамки настоящей главы. Создание прокси-классов подобным образом — исключительно удобная техника, которую можно применять для быстрой генерации клиентского приложения на лету.

Службы WCF с собственным хостингом

До сих пор в этой главе рассматривались службы, развертываемые на веб-серверах. Это позволяет взаимодействовать с ними через Интернет, но для коммуникаций по локальной сети такой способ не слишком эффективен. Во-первых, необходимо программное обеспечение веб-сервера на компьютере, где развернута служба. Во-вторых, архитектура приложения может быть таковой, что существование отдельной от него службы WCF нежелательно.

Взамен можно воспользоваться службой WCF с *собственным хостингом* (self-hosted). Так называется служба, которая существует в процессе, созданном вами, а не в процессе специального хостинг-приложения, такого как веб-сервер. Это означает, например, что для размещения службы может применяться консольное приложение или приложение Windows.

Для этого необходимо работать с классом `System.ServiceMOdel.ServiceHost`. Экземпляр этого класса создается либо с типом службы, которую требуется разместить, либо с экземпляром класса этой службы. Хост службы можно конфигурировать через свойства и методы или же (что рациональнее) через конфигурационный файл. Фактически хост-процессы вроде веб-серверов используют экземпляр `ServiceHost` для осуществления хостинга. Отличие собственного хостинга связано с непосредственным взаимодействием с этим классом. Однако конфигурация, которая помещается в раздел `<system.serviceModel>` файла `app.config` для хост-приложения, использует именно тот же синтаксис, что и разделы конфигурации, которые были показаны в этой главе ранее.

Службу с собственным хостингом можно сделать доступной через любой протокол, хотя в приложениях этого типа обычно применяется привязка TCP или именованный канал. Службы, доступные через HTTP, скорее всего, будут находиться внутри процессов веб-сервера, потому что при этом получается дополнительная функциональность, которую предлагают веб-серверы, такая как средства безопасности и т.д.

Для размещения службы по имени `MyService` применяется следующий код, в котором создается экземпляр `ServiceHost`:

```
ServiceHost host = new ServiceHost(typeof(MyService));
```

Чтобы разместить экземпляр `MyService` по имени `myServiceObject`, можно воспользоваться следующим кодом для создания экземпляра `ServiceHost`:

```
MyService myServiceObject = new MyService();
ServiceHost host = new ServiceHost(myServiceObject);
```

Развертывание экземпляра службы в `ServiceHost` работает, только если служба сконфигурирована так, чтобы вызовы всегда маршрутизировались на один и тот же экземпляр объекта. Для этого необходимо применить к классу службы атрибут `ServiceBehaviorAttribute` и установить свойство `InstanceContextMode` этого атрибута в `InstanceContextMode.Single`.

После создания экземпляра `ServiceHost` можно конфигурировать службу, ее конечные точки и привязки через свойства. В качестве альтернативы, если поместить конфигурацию в файл `.config`, то экземпляр `ServiceHost` будет сконфигурирован автоматически.

Для запуска хостинга службы после настройки экземпляра `ServiceHost` применяется метод `ServiceHost.Open()`. Аналогично, хостинг службы останавливается методом `ServiceHost.Close()`. При первом запуске хостинга привязанной к TCP службы может быть получено предупреждение от службы Windows Firewall (Брандмауэр Windows), поскольку она по умолчанию блокирует порт TCP. Чтобы начать прослушивать порт, понадобится открыть порт TCP службы.

В следующем практическом занятии собственный хостинг используется для предоставления некоторой функциональности приложения WPF через службу WCF.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Службы WCF с собственным хостингом

1. Создайте новое приложение WPF по имени `Ch26Ex03` в каталоге `C:\BeginVCSharp\Chapter26`.
2. Добавьте в проект службу WCF по имени `AppControlService`, используя мастер Add New Item Wizard (Мастер добавления новых элементов).
3. Модифицируйте код в `MainWindow.xaml` следующим образом:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch26Ex03.MainWindow"
  Title="Solar Evolution" Height="450" Width="430"
  Loaded="Window_Loaded" Closing="Window_Closing">
  <Grid Height="400" Width="400" HorizontalAlignment="Center"
  VerticalAlignment="Center">
  <Rectangle Fill="Black" RadiusX="20" RadiusY="20"
    StrokeThickness="10">
    <Rectangle.Stroke>
      <LinearGradientBrush EndPoint="0.358,0.02"
        StartPoint="0.642,0.98">
        <GradientStop Color="#FF121A5D" Offset="0"/>
        <GradientStop Color="#FFB1B9FF" Offset="1"/>
      </LinearGradientBrush>
    </Rectangle.Stroke>
  </Rectangle>
  <Ellipse Name="AnimatableEllipse" Stroke="{x:Null}" Height="0"
    Width="0" HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Color="#FFFFFFFF" Offset="0"/>
        <GradientStop Color="#FFFFFFFF" Offset="1"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
```

```
    <Ellipse.Effect>
      <DropShadowEffect ShadowDepth="0" Color="#FFFFFFFF"
        BlurRadius="50"/>
    </Ellipse.Effect>
  </Ellipse>
  </Grid>
</Window>
```

Фрагмент кода ***Ch26Ex03\MainWindow.xaml***

4. Измените код в `MainWindow.xaml.cs`, как показано ниже:

```
using System.Windows.Shapes;
using System.ServiceModel;
using System.Windows.Media.Animation;
namespace Ch26Ex03
{
  /// <summary>
  /// Логика взаимодействия для MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    private AppControlService service;
    private ServiceHost host;
    public MainWindow()
    {
      InitializeComponent();
    }
    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
      service = new AppControlService(this);
      host = new ServiceHost(service);
      host.Open();
    }
    private void Window_Closing(object sender,
      System.ComponentModel.CancelEventArgs e)
    {
      host.Close();
    }
    internal void SetRadius(double radius, string foreTo,
      TimeSpan duration)
    {
      if (radius > 200)
      {
        radius = 200;
      }
      Color foreToColor = Colors.Red;
      try
      {
        foreToColor =
           (Color)ColorConverter.ConvertFromString(foreTo);
      }
      catch
      {
        // Игнорировать сбой преобразования цвета.
      }
      Duration animationLength = new Duration(duration);
      DoubleAnimation radiusAnimation = new DoubleAnimation(
        radius * 2, animationLength);
      ColorAnimation colorAnimation = new ColorAnimation(
        foreToColor, animationLength);
```

```
        AnimatableEllipse.BeginAnimation(Ellipse.HeightProperty,
          radiusAnimation);
        AnimatableEllipse.BeginAnimation(Ellipse.WidthProperty,
          radiusAnimation);
        ((RadialGradientBrush)AnimatableEllipse.Fill).GradientStops[1]
          .BeginAnimation(GradientStop.ColorProperty, colorAnimation);
      }
    }
  }
```

Фрагмент кода ***Ch26Ex03\MainWindow.xaml.cs***

5. Модифицируйте код в `IAppControlService.cs` следующим образом:

```
[ServiceContract]
public interface IAppControlService
{
  [OperationContract]
  void SetRadius(int radius, string foreTo, int seconds);
}
```

Фрагмент кода ***Ch26Ex03\IAppControlService.cs***

6. Измените код в `AppControlService.cs`, как показано ниже:

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class AppControlService : IAppControlService
{
  private MainWindow hostApp;
  public AppControlService(MainWindow hostApp)
  {
    this.hostApp = hostApp;
  }
  public void SetRadius(int radius, string foreTo, int seconds)
  {
    hostApp.SetRadius(radius, foreTo, new TimeSpan(0, 0, seconds));
  }
}
```

Фрагмент кода ***Ch26Ex03\AppControlService.cs***

7. Модифицируйте код в `app.config` следующим образом:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Ch26Ex03.AppControlService">
        <endpoint address="net.tcp://localhost:8081/AppControlService"
          binding="netTcpBinding"
          contract="Ch26Ex03.IAppControlService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Фрагмент кода ***Ch26Ex03\Web.config***

8. Добавьте к проекту новое консольное приложение по имени `Ch26Ex03CLient`.
9. Щелкните правой кнопкой мыши на решении в окне Solution Explorer и выберите в контекстном меню пункт Set StartUp Projects (Установить стартовые проекты).

10. Выберите переключатель Multiple startup projects (Несколько стартовых проектов) и сделайте оба проекта запускаемыми одновременно, как показано на рис. 26.11.

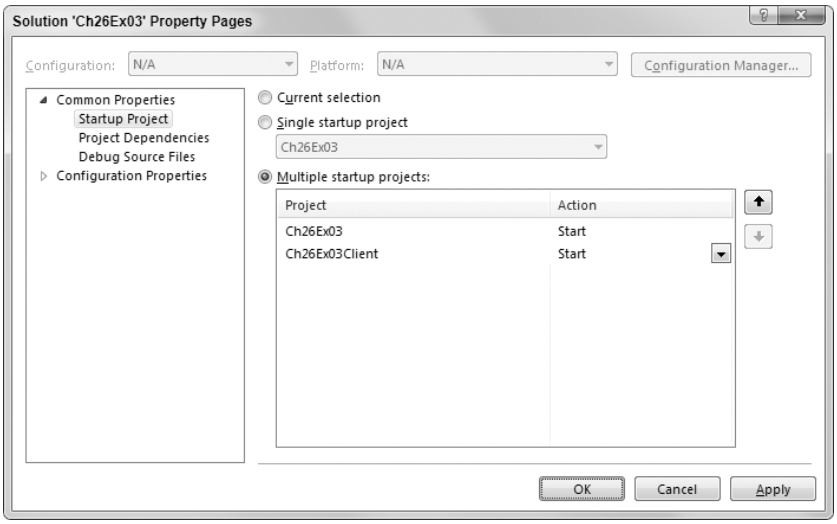


***Рис. 26.11**. Настройка для решения нескольких стартовых проектов*

11. Добавьте к проекту `Cp26Ex03Client` ссылки на `System.Runtime.Serialization ServiceMOde.dll` и `Ch26Ex03`.

12. Модифицируйте код в `Program.cs` следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch26Ex03;
using System.ServiceModel;
namespace Ch26Ex03Client
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Press enter to begin.");
                        // Нажмите Enter, чтобы начать.
      Console.ReadLine();
      Console.WriteLine("Opening channel.");    // открытие канала
      IAppControlService client =
        ChannelFactory<IAppControlService>.CreateChannel(
        new NetTcpBinding(),
        new EndpointAddress(
        "net.tcp://localhost:8081/AppControlService"));
      Console.WriteLine("Creating sun.");       // создание солнца
      client.SetRadius(100, "yellow", 3);
      Console.WriteLine("Press enter to continue.");
                        // Нажмите Enter, чтобы продолжить.
      Console.ReadLine();
      Console.WriteLine("Growing sun to red giant.");
                        // Превращение солнца в красный гигант
```

```
            client.SetRadius(200, "Red", 5);
            Console.WriteLine("Press enter to continue.");
            Console.ReadLine();
            Console.WriteLine("Collapsing sun to neutron star.");
                              // Коллапс солнца в нейтронную звезду
            client.SetRadius(50, "AliceBlue", 2);
            Console.WriteLine("Finished. Press enter to exit.");
                             // Завершено. Нажмите Enter, чтобы выйти.
            Console.ReadLine();
        }
    }
}
```

Фрагмент кода ***Ch26Ex03Client\Program.cs***

13. Запустите решение. Если появится предупреждение, разблокируйте TCP-порт в брандмауэре Windows, чтобы у WCF была возможность прослушивать подключения.
14. Когда отобразятся оба окна — окно Solar Evolution (Солнечная эволюция) и консольное, — нажмите <Enter> в консольном окне. Результат можно видеть на рис. 26.12.

Рис. 26.12*. Приложение в работе*

15. Продолжайте нажимать <Enter> в консольном окне, чтобы приложение двигалось дальше.

Описание работы

В рассмотренном примере в приложение WPF была добавлена служба WCF, которая использовалась для управления анимацией элемента `Ellipse`. Для тестирования службы было написано простое клиентское приложение. Если вы еще не очень знакомы с WPF, не беспокойтесь особенно о коде XAML в данном примере; здесь нас больше интересуют механизмы WCF.

Служба WCF, имеющая имя AppControlService, представляет единственную операцию — SetRadius(), которую вызывает клиент для управления анимацией. Этот метод взаимодействует с идентично именованным методом, определенным в классе Window1 для приложения WPF. Чтобы это работало, службе нужна ссылка на приложение, чтобы можно было развернуть экземпляр объекта в виде службы. Как было описано ранее, это значит, что служба должна использовать атрибут поведения:

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class AppControlService : IAppControlService
{
  ...
}
```

В Window1.xaml.cs экземпляр службы создается в обработчике событий Windows_Loaded(). Этот метод также начинает размещение с создания объекта ServiceHost для службы и вызова его метода Open():

```
public partial class Window1 : Window
{
  private AppControlService service;
  private ServiceHost host;
  ...
  private void Window_Loaded(object sender, RoutedEventArgs e)
  {
    service = new AppControlService(this);
    host = new ServiceHost(service);
    host.Open();
  }
```

Когда приложение закрывается, размещение завершается в обработчике событий Window_Closing().

Конфигурационный файл прост — так и должно быть. В нем определяется единственная конечная точка для службы WCF, которая прослушивает порт 8081 по адресу net.tcp и использует привязку NetTCpBinding:

```
<service name="Ch26Ex03.AppControlService">
  <endpoint address="net.tcp://localhost:8081/AppControlService"
    binding="netTcpBinding"
    contract="Ch26Ex03.IAppControlService" />
</service>
```

Это соответствует следующему коду в клиентском приложении:

```
IAppControlService client =
  ChannelFactory<IAppControlService>.CreateChannel(
    new NetTcpBinding(),
    new EndpointAddress(
      "net.tcp://localhost:8081/AppControlService"));
```

Когда клиент создал прокси-класс, он может вызывать метод SetRadius() с параметрами радиуса, цвета и длительности анимации, и все они передаются приложению WPF через службу. Затем с помощью простого кода в приложении WPF определяется и применяется анимация для изменения размеров и цвета эллипса.

Этот код будет работать и через сеть, если вместо localhost будет указано имя машины, и если сеть пропускает трафик на указанный порт. В качестве альтернативы можно было бы еще более разделить клиента и хост-приложение и подключать их через Интернет. В любом случае службы WCF предоставляют блестящие средства коммуникации, настройка которых не требует больших усилий.

Резюме

В этой главе рассматривались базовые приемы применения служб WCF для взаимодействия между приложениями, процессами и компьютерами. Изучение началось с того, что собой представляет служба WCF, и чем она отличается от веб-службы или реализации Remoting. Вы также ознакомились с концепциями, которые необходимо знать для работы со службами WCF. Затем вы узнали, как программировать службы WCF, как использовать эти службы на стороне клиента, и как развертывать службы WCF различными способами.

В главе был изложен абсолютный минимум того, что нужно знать, чтобы использовать службы WCF в реальных приложениях. Мы лишь кратко коснулись поверхности — в частности, описали конфигурацию и поведения в файле `.config`. Технология WCF допускает интеграцию с расширенными инфраструктурами безопасности, а коммуникации можно настраивать почти любым способом, который только можно вообразить.

За дополнительными сведениями о службах WCF обращайтесь к книге *WCF 4: Windows Communication Foundation и .NET 4 для профессионалов* (Изд-во "Диалектика", 2011 г.). В следующей главе рассматривается последняя из основных технологий, которые появились в .NET 3.5 и были значительно усовершенствованы в .NET 4: Workflow Foundation.

Упражнения

1. Какое из следующих приложений может служить хостом для служб WCF?
   а) Веб-приложения
   б) Приложения Windows Forms
   в) Службы Windows
   г) Приложения COM+
   д) Консольные приложения
2. Какой тип контракта должен быть реализован, если службе WCF необходимо передавать параметр типа `MyClass`? Какие атрибуты понадобятся?
3. Если вы развернете службу WCF в веб-приложении, какое расширение использует базовая точка службы?
4. Для службы WCF с собственным хостингом необходимо конфигурировать службу, устанавливая свойства и вызывая методы класса `ServiceHost`. Верно это или нет?
5. Напишите код для контракта службы `IMusicPLayer` с операциями, определенными для `Play()`, `Stop()` и `GetTRackInformation()`. Используйте однонаправленные методы, где это уместно. Какие еще контракты можно определить для этой службы?

   Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Ключевые концепции
Основы WCF	WCF предоставляет платформу для создания и взаимодействия с удаленными службами. Для этого комбинируются элементы веб-служб и распределенных архитектур с новыми технологиями.
Коммуникационные протоколы	Взаимодействовать со службой WCF можно с помощью любого из нескольких протоколов, включая HTTP и TCP. Это значит, что можно использовать службы, локальные по отношению к клиентскому приложению, или работать с ними через границы машин и сетей. Для этого необходимо обращаться к специфической конечной точке службы через привязку, соответствующую протоколу и средствам, которые нужны. Управлять этими средствами, в том числе использованием состояния сеанса или предоставлением метаданных, можно через поведения. В .NET 4 определено много настроек по умолчанию, облегчающих определение простой службы.
Рабочая нагрузка коммуникаций	Обычно вызовы с ответами от служб WCF закодированы в виде сообщений SOAP. Однако есть и альтернативы, такие как простые сообщения HTTP, а кроме того при необходимости можно определять собственные типы рабочей нагрузки “с нуля”.
Хостинг	Службы WCF могут развертываться в IIS или в службе Windows либо поддерживать собственный хостинг. Использование в качестве хоста IIS делает доступными встроенные в хост возможности, включая средства безопасности и пул приложений. Собственный хостинг является более гибким, но может потребовать дополнительного конфигурирования и кодирования.
Контракты	Интерфейс между службой WCF и клиентским кодом определяется с помощью контрактов. Сами службы, наряду с предоставляемыми ими операциями, определены контрактами службы и операции. Типы данных определяются контрактами данных. Дополнительная настройка коммуникаций достигается контрактами сообщений и сбоев.
Клиентские приложения	Клиентские приложения взаимодействуют со службами WCF посредством прокси-класса. Прокси-классы реализуют интерфейс контракта службы, и все вызовы методов операций этого интерфейса перенаправляются службе. Прокси-класс можно сгенерировать с использованием инструмента Add Service Reference либо создать его программно — через методы фабрики канала. Чтобы коммуникации были успешны, клиент должен быть сконфигурирован в соответствии с конфигурацией сервера.

27

Windows Workflow Foundation

В ЭТОЙ ГЛАВЕ...

- Что собой представляет рабочий поток и как его выполнить
- Что такое действие
- Как создаются специальные действия
- Как отправить электронную почту из действия

Технология Windows Workflow Foundation (WF) впервые появилась в .NET 3.0, а в .NET 3.5 была дополнена полезной функциональностью для облегчения интеграции с Windows Communication Foundation (WCF). В .NET 4 технология Workflow полностью переписана, и хотя базовые концепции остались прежними, реализация целиком изменилась. В этой главе рассматривается версия Windows Workflow Foundation 4 (WF 4).

Рабочий поток упрощенно может быть определен как "коллекция действий", но это определение не вполне удовлетворительно. Возможно, удобнее использовать вместо него аналогию.

При написании программы применяются операторы (вроде `if/else`) и вызовы функций (`Console.WriteLine`), кроме того, наверняка некоторый код выполняется в цикле. Нельзя рассчитывать на знание программирования конечными пользователями, поэтому они говорят вам, чего хотят от системы, а вы пишете код для удовлетворения их нужд.

Теперь представьте на мгновение, что вы можете снабдить конечных пользователей значительно упрощенной средой программирования, оснащенной готовыми операторами и логикой управления потоком, и все, что осталось сделать конечным пользователям — это соединить готовые части вместе, чтобы получить то, что они хотят. Именно для этого и применяется WF 4. Операторы и логика управления называются *действиями* (activity), и эти действия могут быть скомпонованы в рабочий поток.

Пример "Hello World"

В каждой книге по программированию рассматривается пример "Hello World", и эта книга — не исключение. Однако в данном примере вместо традиционного языка программирования будет использоваться WF 4. В следующем практическом занятии мы создадим проект рабочего потока, добавим действие и запустим рабочий поток.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Простое приложение WF 4**

1. В Visual Studio 2010 создайте новый проект консольного приложения рабочего потока (Workflow Console Application). Удостоверьтесь, что в раскрывающемся списке вверху окна выбран вариант .NET Framework 4, как показано на рис. 27.1.

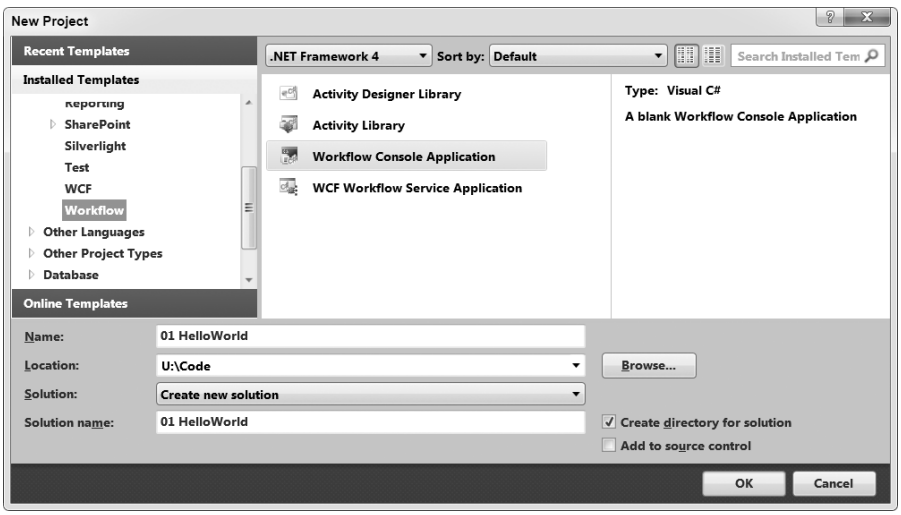


Рис. 27.1. *Создание нового проекта* Workflow Console Application

2. Перетащите действие `WriteLine` из панели инструментов в главную область проектирования.
3. Введите в текстовом поле строку "`Hello Workflow World`" (рис. 27.2).
4. Запустите приложение, чтобы увидеть текст вывода.

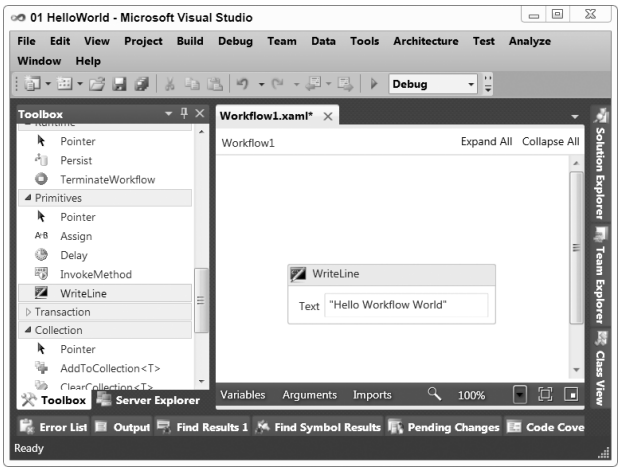


Рис. 27.2. *Ввод текста для действия* `WriteLine`

Описание работы

Когда приложение запускается, оно выполняет действия, содержащиеся в рабочем потоке. В данном примере имеется всего одно такое действие, выводящее некоторый текст; как только действие закончено, завершается и сам рабочий поток, и приложение в целом. Разумеется, реальные приложения содержат гораздо большее количество действий в рабочем потоке, которые решают намного более полезные задачи, чем простой вывод сообщения на консоль.

Рабочие потоки и действия

В предыдущем разделе был показан тривиальный пример рабочего потока, использующего простое действие. Рабочий поток — это коллекция действий, и он определяет последовательность их выполнения. В приведенном примере используется последовательный рабочий поток, который может состоять из нескольких действий, выполняемых в последовательном порядке.

Действие — это единица работы, и доступны два типа действий. Пример действия первого типа вы уже видели — `WriteLine`. Это действие решает только одну задачу. Другим типом является составное действие. Существует несколько примеров таких действий, которые должны быть хорошо знакомы; среди них такие действие `While`, которое на самом деле содержит в себе другие дочерние действия.

Таким образом, рабочий поток подобен программе: он содержит простые действия, которые аналогичны обычным операторам языка программирования, действия по управлению потоком, подобные операторам управления потоком, и выполняет во многом подобно программе.

Если рабочий поток подобен программе, то можно ли здесь создавать собственные функции, как это делается при обычном программировании? Скажем, может понадобиться функция, которая отправляет электронную почту или записывает данные в журнал аудита.

И здесь на помощь приходят специальные действия — вы можете написать такие низкоуровневые области функциональности, а пользователи будут просто подключать их к рабочему потоку.

Технология Windows Workflow Foundation 4 предлагает множество действий, и в следующих разделах мы рассмотрим некоторые из них и покажем, как они могут применяться в рабочем потоке.

Действие `If`

Это действие работает аналогично оператору `if/else` в C#, и при выполнении проверяет условие, а затем решает, по какому пути должен пойти рабочий поток, в зависимости от этого условия.

Действие `If` в рабочем потоке выглядит, как показано на рис. 27.3.

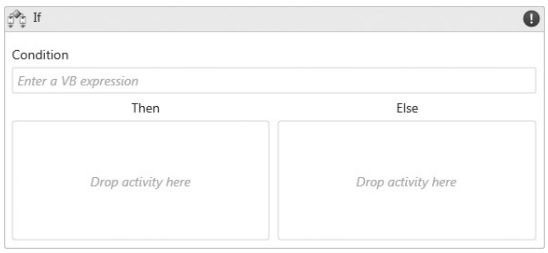


Рис. 27.3. *Действие* `If`

Действие `If` содержит условное выражение, которое оценивается во время выполнения, и места заполнения для действий `Then` и `Else`. Свойство `Condition` — это выражение, которое возвращает булевское значение, так что здесь может применяться любое корректное логическое выражение.

В WF 4 поддерживается механизм выражений, использующий синтаксис Visual Basic. Это может показаться странным для программиста C#, поскольку VB существенно отличается от C#. Однако так обстоят дела сейчас, поэтому для того, чтобы использовать встроенные действия, придется в какой-то мере освоить VB. Просто помните, что писать нужно многословно, пропускать точки с запятой — и все будет хорошо.

Выражение может ссылаться на любые переменные, определенные в рабочем потоке, и имеет доступ к множеству статических классов, определенных в .NET Framework. Так, например, можно определить выражение на основе значения `Environment.Is64BitOperatingSystem`, если это критично для некоторой части рабочего потока. Естественно, можно определять аргументы, передаваемые в рабочий поток, и затем оценивать их в выражении внутри действия `If`. Далее в этой главе мы еще поговорим об аргументах и переменных.

Действие `While`

Действие `While` должно быть знакомым любому программисту. Оно оценивает условие, и пока оно истинно, выполняется тело действия (рис. 27.4).

`While` поддерживает только одно действие внутри тела, но в большинстве случаев в теле цикла необходимо размещать более одного оператора. Это значит, что должен существовать способ для решения данной проблемы, и такой способ есть — действие `Sequence`.

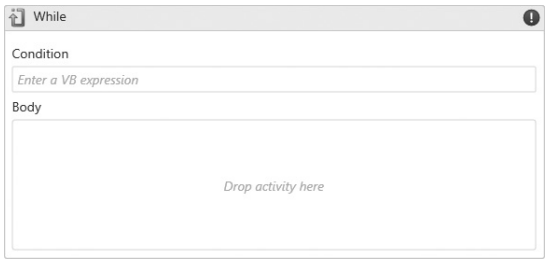


Рис. 27.4. *Действие* `While`

Действие `Sequence`

Это действие позволяет конструировать список других действий, и его выполнение начинается с запуска первого дочернего действия, после чего все остальные действия выполняются по очереди (рис. 27.5).

На рис. 27.5 показано действие `Sequence`, содержащее три дочерних действия — `WriteLine`, `While` и еще одно `WriteLine`. Если запустить это действие, то начальное сообщение будет выведено на консоль, затем выполнится цикл `While`, после чего, наконец, на консоль выводится финальное сообщение. На рис. 27.5 также демонстрируется другое полезное средство визуального конструктора рабочих потоков — он может сворачивать (и разворачивать) действия. В данном случае действие `While` свернуто, так что видно только его имя — кнопка справа от действия с изображением двух стрелок вниз позволяет переключать видимость содержимого действия.

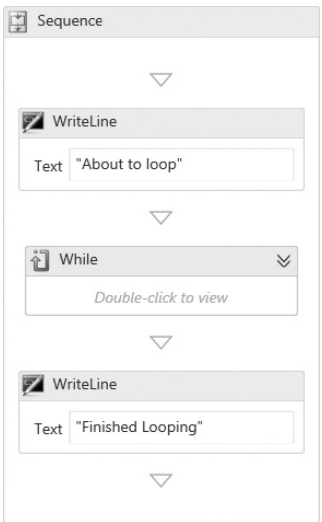


Рис. 27.5. *Действие* `Sequence`

Возможность разворачивания и сворачивания действий очень полезна при проектировании крупных рабочих потоков, поскольку можно скрывать части, с которыми не ведется работа в данный момент, и увеличивать нужные для работы.

Визуальный конструктор рабочих потоков обладает и другими средствами, которые облегчают работу с ним. Например, в правом нижнем углу визуального конструктора находится набор элементов управления, показанных на рис. 27.6.

Эти элементы управления (слева направо) позволяют масштабировать область проектирования до 100%, масштабировать до заданного уровня, располагать текущий рабочий поток так, чтобы он заполнил весь экран, показывать и скрывать обзорное окно, показывающее уменьшенное изображение рабочего потока. Вдобавок двойной щелчок на составном действии вроде `While` или `If` скрывает все действия высшего уровня, оставляя только действие, на котором был совершен щелчок, и его дочерние действия. Навигационная цепочка в верхней части экрана показывает, где вы находитесь (рис. 27.7).

Рис. 27.6. *Элементы управления визуального конструктора рабочих потоков*

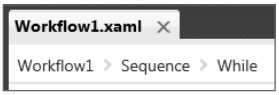


Рис. 27.7. *Навигационная цепочка в визуальном конструкторе рабочих потоков*

Можно щелкать на любом элементе в последовательности, переходя на все более высокий уровень в рабочем потоке, пока не будет достигнут самый верхний уровень.

Аргументы и переменные

Рис. 27.8. *Файл с определением рабочего потока*

В любом нормальном языке программирования можно использовать аргументы для передачи значений в функции (и получать ответы), и внутри функции можно определять и использовать переменные в качестве временного хранилища. Поскольку WF — по сути, язык программирования, те же конструкции доступны и здесь.

Прежде чем переходить к описанию аргументов и переменных, давайте посмотрим, из чего состоит рабочий поток. Если вы откроете окно Solution Explorer в Visual Studio 2010 и посмотрите на решение, созданное в первой части главы, то увидите файл, выделенный на рис. 27.8.

Двойной щелчок на `Workflow1.xaml` приводит к отображению рабочего потока в визуальном конструкторе. Однако это просто обычный файл XML, так что вместо двойного щелчка щелкните на нем правой кнопкой и выберите в контекстном меню пункт Open With (Открыть с помощью). В появившемся диалоговом окне выберите XML Editor (Редактор XML). Файл откроется и будет видна XML-разметка, описывающая рабочий поток:

```
<Activity mc:Ignorable="sap"
  x:Class="_01_HelloWorld.Workflow1"
  mva:VisualBasic.Settings="Assembly references and imported
        namespaces serialized as XML namespaces"
  xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:mv="clr-namespace:Microsoft.VisualBasic;assembly=System"
  xmlns:mva="clr-namespace:Microsoft.
    VisualBasic.Activities;assembly=System.Activities"
  xmlns:s="clr-namespace:System;assembly=mscorlib"
  xmlns:s1="clr-namespace:System;assembly=System"
  xmlns:s2="clr-namespace:System;assembly=System.Xml"
  xmlns:s3="clr-namespace:System;assembly=System.Core"
  xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=System.Activities"
  xmlns:sap="http://schemas.microsoft.com/netfx/2009/xaml/activities/presentation"
  xmlns:scg="clr-namespace:System.Collections.Generic;assembly=System"
  xmlns:scg1="clr-namespace:System.Collections.Generic;assembly=System.ServiceModel"
  xmlns:scg2="clr-namespace:System.Collections.Generic;assembly=System.Core"
  xmlns:scg3="clr-namespace:System.Collections.Generic;assembly=mscorlib"
  xmlns:sd="clr-namespace:System.Data;assembly=System.Data"
  xmlns:sd1="clr-namespace:System.Data;assembly=System.Data.DataSetExtensions"
  xmlns:sl="clr-namespace:System.Linq;assembly=System.Core"
  xmlns:st="clr-namespace:System.Text;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <WriteLine sad:XamlDebuggerXmlReader.FileName=
    "U:\Code\01 HelloWorld\01 HelloWorld\Workflow1.xaml"
    sap:VirtualizedContainerService.HintSize="209.6,200"
    Text="Hello Workflow World" />
</Activity>
```

В этом коде присутствует масса ссылок на пространства имен XML, но основная часть просто отображает действие `WriteLine` со свойством `Text`.

Чтобы создать аргументы, передаваемые в рабочий поток, вы можете использовать конструктор аргументов (Arguments) внутри визуального конструктора рабочего потока. Эта опция находится в нижней левой части поверхности визуального конструктора, как показано на рис. 27.9.

Рис. 27.9. *Опции для вызова других конструкторов*

В следующем практическом занятии мы создадим входной аргумент и переменную, которые применим в простом рабочем потоке.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование аргументов и переменных

1. Создайте проект Workflow Console Application в Visual Studio 2010.
2. После отображения рабочего потока щелкните на кнопке Arguments (Аргументы) и создайте строковый аргумент, как показано на рис. 27.10. Установите имя аргумента — `Name`, направление — `In`, а тип данных — `String`.

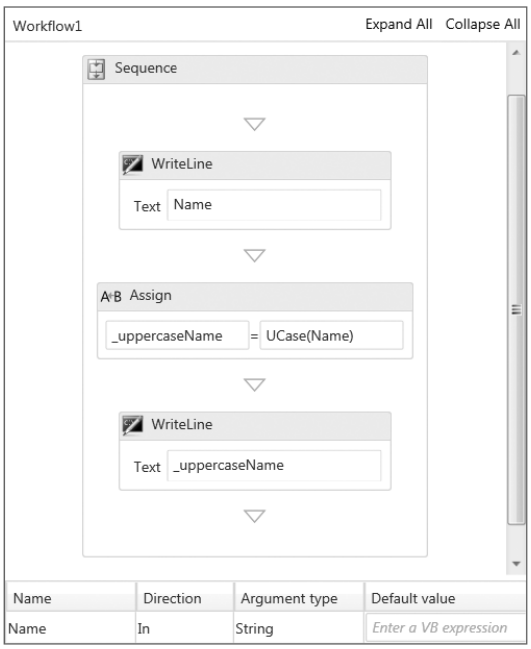


Рис. 27.10*. Создание строкового аргумента*

3. Добавьте к рабочему потоку действие `Sequence`. Добавьте к `Sequence` действие `WriteLine` и введите `Name` в текстовом поле выражения. Обратите внимание на отсутствие кавычек, поскольку теперь это — имя аргумента, а не литеральная строка.
4. Теперь щелкните на кнопке Variables (Переменные) и определите переменную по имени `_uppercaseName` типа `String`. Эта переменная будет использоваться для хранения значения аргумента `Name`, представленного в верхнем регистре.
5. Перетащите действие `Assign` на поверхность визуального конструктора. Введите `_uppercaseName` в левом текстовом поле действия и `UCase(Name)` — в правом поле. Это действие присваивает значение переменной, и значением в данном случае является вызов функции VB, которая преобразует получаемый параметр в строку символов в верхнем регистре.

6. Перетащите действие WriteLine на поверхность визуального конструктора и установите его текст в _uppercaseName.
7. Теперь перейдите к файлу MainProgram.cs, где будет создано значение для аргумента Name и передано в рабочий поток. В этом файле находится следующий код:

```
class Program
{
  static void Main(string[] args)
  {
    WorkflowInvoker.Invoke(new Workflow1());
  }
}
```

8. Модифицируйте приведенный код для приема значения аргумента Name, как показано ниже:

```
class Program
{
  static void Main(string[] args)
  {
    Dictionary<string, object> parms = new Dictionary<string, object>();
    parms.Add("Name", "Morgan");
    WorkflowInvoker.Invoke(new Workflow1(), parms);
  }
}
```

   Если хотите, подставьте свое имя.

9. Скомпонуйте и запустите проект. Должны отобразиться две строки, одна из которых представлена в верхнем регистре.

Описание работы

Во время выполнения это приложение передает строковый параметр в рабочий поток. Данные входного параметра привязываются к действию WriteLine и также действию Assign, где оно преобразуется выражением в версию верхнего регистра, которая затем сохраняется в локальной переменной. Эта переменная используется вторым действием WriteLine.

Если теперь открыть в редакторе XAML результирующий файл Workflow1.xaml, в его верхней части можно увидеть определение аргумента:

```
<x:Members>
  <x:Property Name="Name" Type="InArgument(x:String)" />
</x:Members>
```

Кроме того, имеется также определение переменной внутри последовательности:

```
<Sequence.Variables>
  <Variable x:TypeArguments="x:String" Name="_uppercaseName" />
</Sequence.Variables>
```

Как и переменные в обычной программе, переменные в рабочем потоке имеют область действия, определяющую, где они доступны. Как только завершается действие, в котором определена переменная, эта переменная уничтожается.

В предыдущем примере кода был создан словарь пар “имя/значение”, в который затем было добавлено значение с ключом Name. Значение ключа должно в точности соответствовать определению аргумента; в противном случае значение аргумента не будет установлено корректно, и рабочий поток будет выполняться без соответствующих данных.

Другое поведение функций в нормальном языке программирования связано с возможностью возврата из них значения. Аналогично аргументы можно передавать как в рабочий поток, так и обратно из него. Как и при передаче значений в рабочий поток, любой выходной аргумент возвращается в словаре.

Аргумент включает описание направления, которое может принимать одно из трех значений:

- `In` — аргумент передается в рабочий поток;
- `Out` — аргумент возвращается из рабочего потока;
- `In/Out` — аргумент и передается в рабочий поток, и возвращается из него по завершении.

Только аргументы, определенные как `Out` или `In/Out`, будут возвращаться по завершении рабочего потока. Для чтения значений, возвращенных из рабочего потока, используется следующий код:

```
IDictionary<string,object> returnValues =
   WorkflowInvoker.Invoke(new Workflow1(), parms);
```

Здесь переменной `returnValue` присваивается словарь пар "имя/значение", который будет содержать аргументы `In` и `In/Out`, определенные в рабочем потоке.

В следующем практическом занятии демонстрируется передача аргументов в рабочий поток и обратно.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Возврат аргументов

1. Создайте новый проект Workflow Console Application в Visual Studio 2010.
2. Создайте строковый аргумент рабочего потока и определите его как аргумент `In/Out` по имени `Person`.
3. Перетащите действие `Sequence` в рабочий поток.
4. Перетащите действие `WriteLine` в рабочий поток и установите его текстовое выражение следующим образом:

   ```
   String.Format ( "Person is called : {0}", Person )
   ```

   Это код VB, а не C#; поскольку указано выражение, точка с запятой не нужна.

5. Перетащите действие `Assign` в рабочий поток. Введите в его левой части `Person`, а в правой — следующее выражение:

   ```
   String.Format("You entered the name : {0}", Person)
   ```

   Вы должны в конечном итоге получить рабочий поток, который выглядит так, как показано на рис. 27.11.

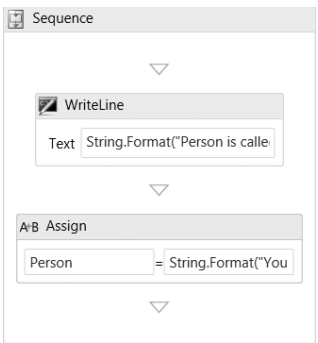


***Рис. 27.11**. Рабочий поток, возвращающий аргумент*

6. Модифицируйте главный файл `program.cs` так, чтобы он передавал аргумент в рабочий поток и выводил значения всех аргументов `Out` или `In/Out`, как показано ниже:

   ```
   Dictionary<string, object> parms = new
   Dictionary<string, object>();
   parms.Add("Person", "Morgan");
   foreach (KeyValuePair<string, object> kvp in
     WorkflowInvoker.Invoke(new Workflow1(), parms))
   {
     Console.WriteLine("{0} = {1}",
        kvp.Key, kvp.Value);
   }
   ```

При выполнении этот рабочий поток должен вывести значение аргумента `Person` рабочего потока, а затем измененное значение должно быть выведено предыдущим кодом. Это доказывает, что аргумент, модифицированный внутри рабочего потока, по его завершении будет передан вызывающему коду.

Описание работы

Когда рабочий поток запускается, он получает входной аргумент. Этот аргумент доступен на протяжении всего времени выполнения рабочего потока, и в рассмотренном примере он также возвращается из рабочего потока по его завершении, поскольку он определен как аргумент `In/Out`.

Специальные действия

До сих пор в этой главе использовались только примеры со встроенными действиями, но WF также допускает написание специальных действий, которые затем применяются точно так же, как встроенные действия.

Ранее в этой главе вы узнали о двух обширных категориях типов действий: одиночные действия и составные действия. В этом разделе мы создадим оба типа.

Действие планируется для выполнения рабочим потоком (или родительским действием), которому оно принадлежит. Что случится дальше — в основном зависит от того, кто написал действие. В случае действия `WriteLine` вполне резонно ожидать наличие вызова `Console.WriteLine` где-то внутри кода действия.

При написании действия обычно переопределяется метод `Execute`, чтобы выполнить собственный код. Этот метод варьируется в зависимости от базового класса, выбранного для действия. Эти базовые классы и их методы выполнения описаны в табл. 27.1.

Таблица 27.1. Методы `Execute` классов действий

Базовый класс	Метод `Execute`
`AsyncCodeActivity`	`IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object)` `void EndExecute(AsyncCodeActivityContext, IAsyncResult)`
`CodeActivity`	`void Execute(CodeActivityContext)`
`NativeActivity`	`void Execute(NativeActivityContext)`
`AsyncCodeActivity<TResult>`	`IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object)` `TResult EndExecute(AsyncCodeActivityContext, IAsyncResult)`
`CodeActivity<TResult>`	`TResult Execute(CodeActivityContext)`
`NativeActivity<TResult>`	`void Execute(NativeActivityContext)`

Простейшим базовым классом для использования является `CodeActivity`. Имеется также его обобщенная версия, принимающая аргумент-тип — она используется в качестве типа возврата от выполнения этого действия. Таким же способом, как рабочий поток может возвращать аргументы, действие может возвращать значение после своего выполнения, и эти данные могут быть привязаны внутри рабочего потока, так что вывод одного действия может формировать ввод для следующего.

Предположим, что в рабочем потоке должно использоваться текущее время. В таком случае можно было бы создать действие, которое вернет значение `DateTime`, и при выпол-

нении получит эту временную метку вызовом `DateTime.Now`. Помимо вывода строки на консоль, это почти самое простое действие, которое только можно представить. В следующем практическом занятии рассматривается создание специального действия.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Написание специального действия

1. Создайте новый проект Workflow Console Application в Visual Studio 2010.
2. Добавьте к решению второй проект, используя для него шаблон Activity Library (Библиотека действий). Будет создано действие по умолчанию (`Activity1.xaml`), которое можно удалить из проекта, потому что здесь оно не используется.
3. Добавьте в библиотеку действий новый класс по имени `Timestamp`. Для этого понадобится следующий код:

```
using System;
using System.Activities;
namespace CustomActivities
{
  public class Timestamp : CodeActivity<DateTime>
  {
    protected override DateTime Execute(CodeActivityContext context)
    {
      return DateTime.Now;
    }
  }
}
```

   Это определит специальное действие и предоставит реализацию соответствующего метода `Execute`, который вернет текущее значение даты/времени.
4. Скомпилируйте решение и затем добавьте ссылку из проекта рабочего потока на проект специального действия. Это позволит использовать специальное действие в проекте рабочего потока и добавит специальное действие в панель инструментов.
5. Отредактируйте главный рабочий поток, перетащив на него последовательное действие. Определите переменную типа `DateTime` в действии `Sequence`. Назовите эту переменную `currentDateTime`.
6. Перетащите действие `Timestamp` и отобразите его свойства. Понадобится изменить свойство `Result` на `currentDateTime`, что присвоит результирующее значение действия этой переменной. На рис. 27.12 показывает значение свойства.

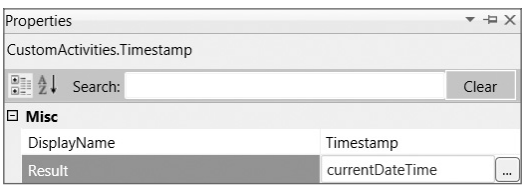


***Рис. 27.12.** Установка значения свойства `Result`*

7. Перетащите действие `WriteLine` и установите его выражение следующим образом:

```
String.Format("The time read from the Timestamp activity is '{0}'",
  currentDateTime)
```

8. Запустите рабочий поток. Вы должны увидеть вывод, представляющий текущую дату и время.

Описание работы

Когда рабочий поток выполняется, он запускает каждое действие по очереди. Если действие унаследовано от `CodeActivity`, то при этом будет вызываться его метод `Execute`. Разрабатываемое в примере действие имеет возвращаемое значение, устанавливаемое внутри метода `Execute` в текущую дату и время. В этом практическом занятии значение действия `Timestamp` сохраняется в переменной рабочего потока, которая затем выводится на консоль с помощью действия `WriteLine`.

Действие `Timestamp` очень простое. Обычно будут создаваться действия, которые выполняют немного больше работы в своих методах `Execute`. Действие обычно представляет собой самодостаточную единицу, подобно функции в традиционном языке программирования. Обычно функции имеют один или более аргументов, и эти аргументы обычно передаются в функцию в виде параметров. Однако иногда функция получает данные через текущий контекст приложения.

Хороший пример можно найти в ASP.NET. Статическое свойство, доступное как `HttpContext.Current`, предоставляет доступ к различным свойствам, таким как текущее состояние приложения, `HttpRequest` и `HttpResponse`. Этот объект определен конвейером обработки ASP.NET и потому доступен любому объекту, вызываемому в этом конвейере.

Аналогичное средство имеется и в WF и оно основано на концепции *расширений*.

Расширения рабочего потока

Расширение (extension) — это просто объект, который должен быть доступным из действия внутри метода `Execute`. Обычно определяется интерфейс расширения, и действие кодируется согласно этому интерфейсу. Это позволяет заменять реализацию расширения без необходимости в перекодировании всего действия.

В качестве примера рассмотрим действие, отправляющее сообщение электронной почты. Поставщик электронной почты может быть жестко закодирован внутри самого действия, но тогда работа будет осуществляться только с ним. В этом случае лучше определить интерфейс, используемый действием, и предоставить несколько реализаций этого интерфейса: один — для отправки почты через Outlook, другой — с использованием Micrtosoft Exchange и т.д. Короче говоря, список поставщиков электронной почты может быть расширен без изменения действия.

Для того чтобы действие по отправке электронной почты могло использовать любой из поставщиков, понадобится возможность определения аргументов, передаваемых действию. Вы определенно не захотите жестко кодировать получателя электронной почты, тему или тело сообщений. Так же как это делалось с действием `WriteLine`, нужно будет определить свойства, которые могут быть установлены в действии. Для этого необходимо использовать классы, производные от `Argument`, такие как `InArgument` и `OutArgument`.

Эти классы аргументов служат для помещения свойств в действие. Причина применения этих типов аргументов связана с тем, как рабочий поток сохраняет свое состояние в процессе выполнения, о чем будет сказано далее в этой главе. А пока давайте создадим интерфейс и специальное действие.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ **Определение интерфейса `ISendEmail` и действия**

1. Создайте новый проект Workflow Console Application в Visual Studio 2010. Назовите его `SharedInterfaces`.
2. Добавьте интерфейс к библиотеке по имени `ISendEmail`, как показано ниже:

```
/// <summary>
/// Интерфейс, используемый действием SendEmail для отправки электронной почты
/// </summary>
public interface ISendEmail
{
  /// <summary>
  /// Отправляет сообщение электронной почты
  /// </summary>
  /// <param name="sender">Отправитель сообщения</param>
  /// <param name="recipient">Получатель сообщения</param>
  /// <param name="subject">Тема сообщения</param>
  /// <param name="body">Тело сообщения</param>
  void SendEmail(string sender, string recipient, string subject, string body);
}
```

3. Теперь добавьте к решению второй проект. На этот раз выберите шаблон Activity Library и назовите его `CustomActivities`. Удалите автоматически созданный файл `Activity1.xaml`.

4. Добавьте в проект новый класс по имени `SendEmail`. Определите его, как показано в следующем примере. Чтобы включить определение интерфейса `IsendEmail`, понадобится ссылка на проект `SharedInterfaces` из библиотеки действий:

```
public class SendEmail : NativeActivity
{
  public InArgument<string> Sender { get; set; }
  public InArgument<string> Recipient { get; set; }
  public InArgument<string> Subject { get; set; }
  public InArgument<string> Body { get; set; }
  protected override void Execute(NativeActivityContext context)
  {
    context.GetExtension<ISendEmail>().SendEmail
       (Sender.Get(context),Recipient.Get(context),
       Subject.Get(context), Body.Get(context));
  }
}
```

Действие определяет четыре входных аргумента и использует их внутри метода `Execute`.

Описание работы

Когда действие выполняется, оно просто ищет расширение `ISendEmail` и вызывает его метод `SendEmail`. Для отправки электронной почты требуется еще немало функциональности, которая будет рассматриваться в последующих практических занятиях.

При определении аргументов должны использоваться обобщенные классы `InArgument<>`, `OutArgument<>` или `InOutArgument<>`. Внутри метода `Execute` значение этих аргументов извлекается из текущего контекста выполнения с помощью несколько странного синтаксиса `Argument.Get(context)`. Это связано с тем, как данные хранятся в рабочем потоке.

В традиционном классе с обычными свойствами .NET данные для этого класса сохраняются внутри экземпляра объекта. Это делает их невидимыми для внешнего вызывающего контекста; в случаях, когда им является действие, это означало бы сохранение экземпляра рабочего потока на диск, и при этом каждое действие должно было бы полностью сериализироваться, прежде чем рабочий поток можно было сохранить на диск.

Так работала технология WF 3.x, и для сохранения некоторых рабочих потоков при этом требовался значительный объем дискового пространства. В новой модели, представленной в WF 4, сохраняются только изменяемые данные, поскольку объект контекста, передаваемый действию, может сохранять закладки на действительных значениях этих ар-

гументов. В действии SendEmail данные читаются из контекста только с использованием метода Get(), так что состояние рабочего потока поддерживается при выполнении этого действия. Например, если вызывается метод Set(), который изменяет значение аргумента, то логика выполнения рабочего потока должна будет установить флаг, указывающий, что что-то изменилось, позволяя ему сохранить на диск только эти изменения. Это обеспечивает гораздо более высокую производительность в WF 4 и потенциально намного меньший расход дискового пространства.

В дополнение к поддержке множества аргументов, объект контекста также включает коллекцию расширений. В предыдущем коде можно извлечь интерфейс ISendEmail из объекта контекста, вызывая обобщенный метод GetExtension<>. Необходимо передать тип запрашиваемого интерфейса, и логика поиска внутри этого метода вернет в код экземпляр расширения, на котором можно вызвать метод SendEmail.

Следующий шаг, рассматриваемый в приведенном ниже практическом занятии, состоит в добавлении расширения к рабочему потоку, и для того, чтобы сделать это, необходимо использовать другой класс из сборок рабочих потоков — WorkflowApplication.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Использование класса WorkflowApplication

1. Добавьте к решению третий проект. Это должен быть проект Workflow Console Application. Установите его в качестве стартового проекта.
2. Добавьте ссылки на сборки SharedInterfaces и CustomActivities.
3. Добавьте реализацию интерфейса ISendEmail, как показано ниже (она не будет отправлять электронную почту, но, по крайней мере, выведет данные на консоль):

```
public class ConsoleSendEmail : ISendEmail
{
  public void SendEmail(string sender, string recipient, string subject,
    string body)
  {
    Console.WriteLine("Email to: {0}", recipient);
    Console.WriteLine(" from: {0}", sender);
    Console.WriteLine(" subject: {0}", subject);
    Console.WriteLine(" body: {0}", body);
  }
}
```

4. Добавьте действие SendEmail к рабочему потоку и установите все его свойства. Чтобы отобразить таблицу свойств для выбранного действия, нажмите <F4>.
5. Модифицируйте файл Program.cs и воспользуйтесь классом WorkflowApplication для выполнения рабочего потока. Этот класс позволяет добавлять расширения, тогда как WorkflowInvoker — нет.

```
class Program
{
  static void Main(string[] args)
  {
    WorkflowApplication app = new WorkflowApplication(new Workflow1());
    app.Extensions.Add(new ConsoleSendEmail());
    ManualResetEvent finished = new ManualResetEvent(false);
    app.Completed = (completedArgs) => { finished.Set(); };
    app.Run();
    finished.WaitOne();
  }
}
```

Здесь создан экземпляр WorkflowApplication, к которому добавлено расширение ConsoleSendEmail. Затем создан экземпляр ManualResultEvent и присоединен к событию Completed, которое инициируется по завершении рабочего потока. Затем рабочий поток запускается вызовом метода Run, и вы ожидается его завершение за счет ожидания события. Скомпилировав и запустив программу, вы должны будете увидеть некоторый вывод в окне консоли, соответствующий значению свойств, установленных действием SendEmail.

Описание работы

Когда выполняется приложение рабочего потока, добавленные к приложению расширения сохраняются в коллекции внутри объекта WorkflowApplication. Этот объект планирует выполнение каждого действия, и когда действие выполняется, создается объект контекста и передается методу Execute.

Если бы наследование осуществлялось от CodeActivity, то объект контекста, передаваемый методу Execute, не включал бы доступ к каким-либо расширениям — действие кода имеет очень ограниченный доступ к контекстной информации, что определено проектным решением.

При вызове Run на приложении рабочего потока используется поток (thread) из пула потоков процесса для выполнения рабочего потока (workflow), что позволяет коду продолжить работу, пока рабочий поток выполняется в фоновом режиме. Предыдущий код синхронизирует главное приложение с завершением рабочего потока, используя ManualResetEvent и устанавливая его внутри обработчика события Completed.

Запустив этот код и удостоверившись, что действие SendEmail работает, теперь можно создать реальную реализацию ISendEmail, используя классы из пространства имен System.Net.Mail. Отличным источником информации по этому пространству имен послужит сайт www.systemnetmail.com.

Альтернативой для отправки электронной почты является использование Outlook, что демонстрируется в следующем примере.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Отправка электронной почты с использованием Outlook

1. Добавьте к решению третий проект. Это должен быть проект Workflow Console Application. Установите его в качестве стартового проекта.
2. Добавьте ссылку на объектную модель Outlook. Для этого откройте диалоговое окно Add Reference (Добавление ссылки) и перейдите на вкладку COM. Затем прокрутите список вниз, пока не найдете элемент Microsoft Outlook Object Library (Библиотека объектов Microsoft Outlook). На машине, используемой для выполнения этого примера, установлен пакет Microsoft Office 2007, с внутренним номером версии 12.0, что и видно в диалоговом окне, показанном на рис. 27.13.
3. Создайте новый класс по имени OutlookSendEmail и введите следующий код:

```
public class OutlookSendEmail : ISendEmail
{
  public void SendEmail(string sender, string recipient, string subject,
    string body)
  {
    Application app = new Application();
    var mapi = app.GetNamespace("MAPI");
    mapi.Logon(ShowDialog: false, NewSession: false);
    var outbox = mapi.GetDefaultFolder(OlDefaultFolders.olFolderOutbox);
    MailItem email = app.CreateItem(OlItemType.olMailItem);
    email.To = recipient;
```

```
        email.Subject = subject;
        email.Body = body;
        email.Send();
      }
    }
```

Обратите внимание, что в этом случае отправитель не указывается, поскольку электронная почта будет отправлена с использованием профиля текущего зарегистрированного в системе пользователя.

4. Измените файл `Program.cs` для использования этого класса в качестве расширения электронной почты:

   ```
   app.Extensions.Add(new OutlookSendEmail());
   ```

   Расширение `ConsoleSendEmail` понадобится удалить либо просто закомментировать.

5. Запустите программу (убедившись также, что Outlook работает). Если вы заглянете на вкладку Sent Items (Отправленные), то должны будете увидеть там автоматически сгенерированное сообщение. Если это так, значит, вы только что отправили первое автоматизированное сообщение электронной почты.

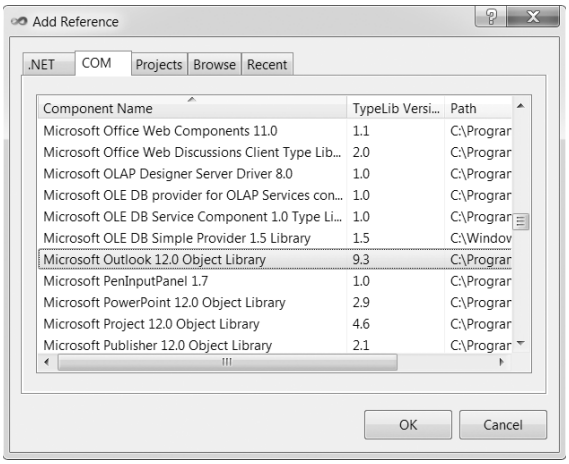


***Рис. 27.13**. Добавление в проект библиотеки объектов Microsoft Outlook*

Описание работы

Класс `OutlookSendEmail` использует объектную модель Outlook для создания нового сообщения электронной почты — в частности, класс `MailItem`. Чтобы отправить электронную почту через Outlook, нужно сконструировать экземпляр объекта `Application` и затем получить ссылку на пространство имен `MAPI`.

Если Outlook пока еще не запущен, в вызове `Login` понадобится указать имя пользователя и пароль. Если же он запущен, эти параметры можно опустить, и в этом случае будет применен профиль текущего зарегистрированного пользователя.

После создания `MailItem` можно указать получателя, установив для этого свойство `To`. Чтобы отправить письмо нескольким адресатам, необходимые адреса электронной почты должны быть перечислены через точку с запятой. Затем потребуется указать тело (`Body`) и тему (`Subject`) сообщения электронной почты и, наконец, вызвать `Send()`. Для отправки форматированного сообщения для установки тела должно использоваться свойство `HTMLBody`.

Если сообщение в папке "Отправленные" отсутствует, или же во время выполнения возникает исключение, необходимо разобраться с причинами. Для этого можно добавить некоторый дополнительный код в файл `Program.cs`, как показано в следующем практическом занятии.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Обработка ошибок в рабочих потоках

1. Откройте проект из предыдущего примера и модифицируйте файл `Program.cs` так, чтобы он выглядел следующим образом:

```
static void Main(string[] args)
{
  WorkflowApplication app = new WorkflowApplication(new Workflow1());
  app.OnUnhandledException = (e) =>
    {
      return UnhandledExceptionAction.Abort;
    };
  app.Extensions.Add(new OutlookSendEmail());
  ManualResetEvent finished = new ManualResetEvent(false);
  app.Completed = (completedArgs) => { finished.Set(); };
  app.Aborted = (abortedEventArgs) =>
    {
      Console.WriteLine("Workflow Aborted.\r\n{0}",abortedEventArgs.Reason);
      finished.Set();
    };
  app.Run();
  finished.WaitOne();
}
```

   Добавленный код выделен.

2. Запустите приложение. Если возникнет исключение, информация о нем выводится на консоль после сообщения, начинающегося с "Workflow Aborted".

Описание работы

Когда в рабочем потоке возникает необработанное исключение, то первое, что вызывается при этом — это делегат `OnUnhandledException`. Здесь можно выбрать действие, которое следует предпринять: `Abort`, `Cancel` или `Terminate`. Делегат получает экземпляр исключения, так что в зависимости от типа исключения можно решить, какое действие предпринять.

Если выбрано действие `Abort`, за ним вызывается делегат `Aborted`. По умолчанию это будет `Terminate`, которое прервет рабочий поток.

Проверка достоверности действия

Многие действия не могут функционировать без определенных аргументов, но в настоящее время нет возможности пометить определенный аргумент как обязательный. Вы могли заметить сообщение об ошибке, отображающееся в верхней части визуального конструктора рабочего потока при использовании некоторых стандартных действий, поскольку у них предусмотрены обязательные аргументы.

Для пометки свойства как обязательного можно использовать атрибут `[RequiredArgument]` при определении аргумента. После добавления его к аргументу справа от действия появляется маленький восклицательный знак красного цвета, как показано на рис. 27.14.

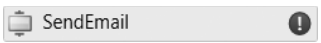


***Рис. 27.14**. Действие имеет обязательные аргументы*

Это указывает на то, что одно или более свойств содержат ошибки. Если навести курсор мыши на этот знак, будет отображена всплывающая подсказка с описанием ошибки. В следующем практическом занятии мы изменим действие SendMail и пометим все аргументы, кроме Sender, как обязательные.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Пометка аргументов как обязательных

1. Откройте файл SendEmail.cs и внесите следующие изменения:

```
public class SendEmail : NativeActivity
{
  public InArgument<string> Sender { get; set; }
  [RequiredArgument]
  public InArgument<string> Recipient { get; set; }
  [RequiredArgument]
  public InArgument<string> Subject { get; set; }
  [RequiredArgument]
  public InArgument<string> Body { get; set; }
  protected override void Execute(NativeActivityContext context)
  {
    context.GetExtension<ISendEmail>().SendEmail(Sender.Get(context),
      Recipient.
    Get(context), Subject.Get(context), Body.Get(context));
  }
}
```

2. Скомпилируйте приложение.
3. Откройте файл Workflow1.xaml и отобразите свойства действия SendEmail. Измените одно из свойств, помеченных атрибутом [RequiredArgument], и с помощью клавиши табуляции выйдите из текстового поля. Вы должны увидеть метку ошибки (восклицательный знак), а при наведении курсора — ее описание.
4. Введите значение обязательного аргумента и снова выйдите из поля табуляцией. Сообщение об ошибке должно исчезнуть.

Описание работы

Рабочий поток был спроектирован для инспектирования действий и поиска атрибута RequiredArgument. Если находятся свойства с таким атрибутом, которые не имеют заданного значения, то класс визуального конструктора помечает действие меткой ошибки.

Специальное действие почти готово. Последняя задача связана с созданием специального визуального конструктора, который обеспечит высокоуровневую визуализацию действия.

Визуальные конструкторы действий

Когда действие перетаскивается на поверхность проектирования, его визуальное представление обеспечивается визуальным конструктором. Обычно это был класс Windows Forms, но в Visual Studio 2010 теперь можно использовать XAML для определения визуальных конструкторов действий.

Язык XAML подробно обсуждался в главе 22, так что мы не станем здесь повторяться. Взамен в данном разделе мы сосредоточим внимание на том, что касается специальных действий.

Класс визуального конструктора для действия обычно создается в отдельной сборке, т.к. он нужен только во время проектирования, а не при выполнении действия. Visual Studio 2010 включает тип проекта Activity Designer Library (Библиотека визуальных конст-

рукторов действий), который предлагает достаточно первоначальной функциональности, и именно он будет применяться в следующем примере.

В дополнение к обеспечению визуального представления действия визуальный конструктор может также использоваться для представления полей ввода данных внутри себя. Без визуального конструктора все свойства действия должны быть установлены внутри сетки свойств; в специальном визуальном конструкторе можно поместить некоторые свойства на саму поверхность проектирования. Это поможет обеспечить для пользователей замечательный внешний вид действия во время проектирования.

В следующем практическом занятии к действию `SendMail` будет добавлен специальный визуальный конструктор.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ

Добавление визуального конструктора действия

1. Откройте предыдущее решение и добавьте в него новый проект Activity Designer Library. Назовите его `CustomActivities.Design`.

2. Будет создан пустой визуальный конструктор по имени `ActivityDesigner1`. Его можно переименовать либо добавить новый визуальный конструктор по имени `SendEmailDesigner`. В любом случае должен быть получен визуальный конструктор, чье имя похоже на имя действия, с которым он будет использоваться.

3. Созданная XAML-разметка по умолчанию предоставляет пустую поверхность проектирования, на которую необходимо добавить некоторые текстовые поля и метки. Добавьте показанную ниже разметку в XAML-файл визуального конструктора — она определит набор столбцов и строк, в которые будут помещены элементы дизайна:

```
<sap:ActivityDesigner x:Class="CustomActivities.Design.SendEmailDesigner"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sap="clr-namespace:System.Activities.Presentation;
  assembly=System.Activities.Presentation"
  xmlns:sapv="clr-namespace:System.Activities.Presentation.View;
  assembly=System.Activities.Presentation">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
  </Grid>
</sap:ActivityDesigner>
```

4. Добавьте элементы, которые будут использованы на экране для приема пользовательского ввода:

```
  </Grid.RowDefinitions>
  <TextBlock Text="Recipient"/>
  <TextBox Text="{Binding ModelItem.Recipient}" Grid.Column="1"/>
  <TextBlock Text="Subject" Grid.Row="1"/>
  <TextBox Text="{Binding ModelItem.Subject}" Grid.Row="1" Grid.Column="1"/>
  <TextBlock Text="Body" Grid.Row="2"/>
  <TextBox Text="{Binding ModelItem.Body}" Grid.Row="2" Grid.Column="1"/>
</Grid>
```

Эти элементы определяют набор меток и текстовых полей, привязанных к данным лежащего в основе действия — префикс `ModelItem` является синонимом действительного действия.

5. Визуальный конструктор должен быть ассоциирован с действием. Проще всего это сделать с помощью атрибута `Designer`. Добавьте следующий код в начало действия `SendEmail`:

```
using System.ComponentModel;
namespace CustomActivities
{
  [Designer("CustomActivities.Design.SendEmailDesigner,
  CustomActivities.Design")]
  public class SendEmail : NativeActivity
  {
```

Атрибут `Designer` читается Visual Studio и применяется для определения визуального конструктора, ассоциированного с действием, а также сборки, содержащей визуальный конструктор. Использованная выше строка — это имя типа (`TypeName`) визуального конструктора, и оно обычно вводится как строка, чтобы избежать необходимости ссылок между сборкой визуального конструктора и сборкой действия.

6. Добавьте ссылку на сборку `PresentationCore` из главной сборки рабочего потока. Скомпилировав решение и открыв рабочий поток, содержащий действие `SendEmail`, вы должны увидеть нечто подобное показанному на рис. 27.15.

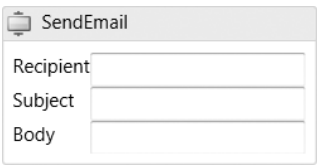


Рис. 27.15. *Визуальный конструктор действия* `SendEmail`

7. Этот дизайн функционален, но не слишком привлекателен, и он может выиграть от добавления некоторых отступов вокруг полей. Измените XAML-разметку, как показано в ниже, и вы получите более привлекательный результат. Разумеется, можно также добавить цвета и графику, чтобы еще больше улучшить впечатление от дизайна.

```
<Grid>
   <Grid.Resources>
     <Style TargetType="TextBlock">
       <Setter Property="Margin" Value="0,2,4,2"/>
       <Setter Property="VerticalAlignment" Value="Center"/>
     </Style>
       <Style TargetType="TextBox">
       <Setter Property="Margin" Value="0,2,0,2"/>
     </Style>
   </Grid.Resources>
   <Grid.ColumnDefinitions>
```

Эти ресурсы определяют стили, ассоциированные с блоками текста и текстовыми полями. Здесь стили просто применяют унифицированные поля и выравнивание, чтобы действие лучше выглядело на экране.

Описание работы

Среда Visual Studio использует атрибут `Designer` для нахождения класса, ассоциированного с действием. После обнаружения этот класс используется при отображении действия на экране.

Нередко в XAML-разметке используется привязка данных для связывания визуального класса с фоновым классом; в рассматриваемом случае визуальным классом является визуальный конструктор, а фоновым классом — действие.

Резюме

В этой главе вы ознакомились с технологией Windows Workflow Foundation 4. В частности, были описаны следующие темы.

- Что собой представляет рабочий поток и как его выполнить.
- Как использовать некоторые из встроенных действий.
- Как создавать собственные действия.

Упражнения

1. Как бы вы создали составное действие?
2. Можно ли предоставить рабочий поток через WCF? Если да, то как?
3. Как обеспечить возможность запуска рабочего потока с места, где он был остановлен?

Решения упражнений приведены в приложении А.

Что вы узнали в этой главе

Тема	Ключевые концепции
Основы рабочих потоков	Рабочие потоки состоят из действий, и каждое действие подобно оператору в традиционном языке программирования. Имеется возможность писать собственные действия, и обычно рабочие потоки состоят из ряда встроенных действий и нескольких специальных.
Действие `If`	Действие `If` может быть использовано в рабочем потоке для оценки выражения и выбора одного из двух путей выполнения. Выражение может быть простым или сложным, и при необходимости может ссылаться на переменные и аргументы.
Действие `While`	Действие `While` позволяет создавать циклы в рабочем потоке. Условие цикла — выражение, а действие в его теле — единственное дочернее действие, которое обычно будет последовательностью, чтобы можно было добавлять несколько других действий в каждую итерацию цикла.
Действие `Sequence`	Действие `Sequence` позволяет выполнять множество дочерних действий в строгом порядке — сверху вниз.
Аргументы и переменные	Аргументы можно передавать в рабочий поток и из него, а внутри рабочего потока определять переменные, имеющие глобальную или локальную область действия. Аргументы определяются типом данных, таким как `String` или `Int32`, а также направлением. Переменные подчиняются тем же правилам, что и в традиционном языке программирования.
Расширения рабочего потока	Расширения могут использоваться для изменения поведения во время выполнения, без необходимости в модификации самого рабочего потока. Расширение обычно создается как интерфейс и реализация этого интерфейса.
Проверка достоверности действия	Некоторые свойства действия могут быть определены как обязательные. Визуальные значки в пользовательском интерфейсе у каждого из незаполненных свойств позволяют видеть, какие свойства должны обязательно получить значения.
Визуальные конструкторы действий	Для дополнения пользовательского интерфейса действия может применяться визуальный конструктор, который облегчает конечному пользователю работу с действием. Визуальный конструктор — это XAML-разметка, и для отображения пользовательского интерфейса специального действия допускается создавать практически любую разметку.

Приложение A

Решения упражнений

В главах 1 и 2 упражнений нет.

Глава 3

Упражнение 1

```
super.smashing.great
```

Упражнение 2

б), так как начинается с числа, и д), поскольку содержит точку.

Упражнение 3

Нет, теоретического предела размера строки, которая может содержаться в переменной `string`, не существует.

Упражнение 4

Операции * и / здесь имеют наивысший приоритет, за ними идут +, << и, наконец, +=. Приоритеты могут быть проиллюстрированы с помощью скобок:

```
resultVar += (((var1 * var2) + var3) << (var4 / var5));
```

Упражнение 5

```
static void Main(string[] args)
{
  int firstNumber, secondNumber, thirdNumber, fourthNumber;
  Console.WriteLine("Введите число:");
  firstNumber = Convert.ToInt32(Console.ReadLine());
  Console.WriteLine("Введите еще одно число:");
  secondNumber = Convert.ToInt32(Console.ReadLine());
  Console.WriteLine("Введите еще одно число:");
```

```
    thirdNumber = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Введите еще одно число:");
    fourthNumber = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Результат перемножения {0}, {1}, {2} и {3} равен {4}.",
              firstNumber, secondNumber, thirdNumber, fourthNumber,
              firstNumber * secondNumber * thirdNumber * fourthNumber);
}
```

Обратите внимание, что здесь используется метод `Convert.ToInt32()`, не описанный в этой главе.

Глава 4

Упражнение 1

```
(var1 > 10) ^ (var2 > 10)
```

Упражнение 2

```
static void Main(string[] args)
{
  bool numbersOK = false;
  double var1, var2;
  var1 = 0;
  var2 = 0;
  while (!numbersOK)
  {
    Console.WriteLine("Введите число:");
    var1 = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Введите еще одно число:");
    var2 = Convert.ToDouble(Console.ReadLine());
    if ((var1 > 10) ^ (var2 > 10))
    {
      numbersOK = true;
    }
    else
    {
      if ((var1 <= 10) && (var2 <= 10))
      {
        numbersOK = true;
      }
      else
      {
        Console.WriteLine("Только одно число может быть больше 10.");
      }
    }
  }
  Console.WriteLine("Вы ввели {0} и {1}.", var1, var2);
}
```

Обратите внимание, что это можно сделать лучше с использованием другой логики, например:

```
static void Main(string[] args)
{
  bool numbersOK = false;
  double var1, var2;
  var1 = 0;
  var2 = 0;
  while (!numbersOK)
  {
```

```
      Console.WriteLine("Введите число:");
      var1 = Convert.ToDouble(Console.ReadLine());
      Console.WriteLine("Введите еще одно число:");
      var2 = Convert.ToDouble(Console.ReadLine());
      if ((var1 > 10) && (var2 > 10))
      {
        Console.WriteLine("Только одно число может быть больше 10.");
      }
      else
      {
        numbersOK = true;
      }
    }
    Console.WriteLine("Вы ввели {0} и {1}.", var1, var2);
}
```

Упражнение 3

Код должен выглядеть так:

```
int i;
for (i = 1; i <= 10; i++)
{
  if ((i % 2) == 0)
    continue;
  Console.WriteLine(i);
}
```

Использование операции присваивания = вместо булевской операции == является весьма распространенной ошибкой.

Упражнение 4

```
static void Main(string[] args)
{
  double realCoord, imagCoord;
  double realMax = 1.77;
  double realMin = -0.6;
  double imagMax = -1.2;
  double imagMin = 1.2;
  double realStep;
  double imagStep;
  double realTemp, imagTemp, realTemp2, arg;
  int iterations;
  while (true)
  {
    realStep = (realMax - realMin) / 79;
    imagStep = (imagMax - imagMin) / 48;
    for (imagCoord = imagMin; imagCoord >= imagMax;
      imagCoord += imagStep)
    {
      for (realCoord = realMin; realCoord <= realMax;
      realCoord += realStep)
      {
        iterations = 0;
        realTemp = realCoord;
        imagTemp = imagCoord;
        arg = (realCoord * realCoord) + (imagCoord * imagCoord);
        while ((arg < 4) && (iterations < 40))
        {
          realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
          imagTemp = (2 * realTemp * imagTemp) - imagCoord;
```

```
            realTemp = realTemp2;
            arg = (realTemp * realTemp) + (imagTemp * imagTemp);
            iterations += 1;
          }
          switch (iterations % 4)
          {
            case 0:
              Console.Write(".");
              break;
            case 1:
              Console.Write("o");
              break;
            case 2:
              Console.Write("O");
              break;
            case 3:
              Console.Write("@");
              break;
          }
        }
        Console.Write("\n");
      }
      Console.WriteLine("Текущие пределы:");
      Console.WriteLine("realCoord: от {0} до {1}", realMin, realMax);
      Console.WriteLine("imagCoord: от {0} до {1}", imagMin, imagMax);
      Console.WriteLine("Введите новые ограничения:");
      Console.WriteLine("realCoord: от:");
      realMin = Convert.ToDouble(Console.ReadLine());
      Console.WriteLine("realCoord: до:");
      realMax = Convert.ToDouble(Console.ReadLine());
      Console.WriteLine("imagCoord: от:");
      imagMin = Convert.ToDouble(Console.ReadLine());
      Console.WriteLine("imagCoord: до:");
      imagMax = Convert.ToDouble(Console.ReadLine());
    }
  }
```

Глава 5

Упражнение 1

Преобразования а) и в) нельзя выполнить неявно.

Упражнение 2

```
enum color : short
{
  Red, Orange, Yellow, Green, Blue, Indigo, Violet, Black, White
}
```

Да, поскольку тип `byte` может содержать числа от 0 до 255, поэтому перечисления на основе `byte` могут содержать 256 элементов с индивидуальными значениями или больше, если присутствуют дубликаты.

Упражнение 3

```
static void Main(string[] args)
{
  imagNum coord, temp;
  double realTemp2, arg;
  int iterations;
```

```
    for (coord.imag = 1.2; coord.imag >= -1.2; coord.imag -= 0.05)
    {
      for (coord.real = -0.6; coord.real <= 1.77; coord.real += 0.03)
      {
        iterations = 0;
        temp.real = coord.real;
        temp.imag = coord.imag;
        arg = (coord.real * coord.real) + (coord.imag * coord.imag);
        while ((arg < 4) && (iterations < 40))
        {
          realTemp2 = (temp.real * temp.real) - (temp.imag * temp.imag)
            - coord.real;
          temp.imag = (2 * temp.real * temp.imag) - coord.imag;
          temp.real = realTemp2;
          arg = (temp.real * temp.real) + (temp.imag * temp.imag);
          iterations += 1;
        }
        switch (iterations % 4)
        {
          case 0:
            Console.Write(".");
            break;
          case 1:
            Console.Write("o");
            break;
          case 2:
            Console.Write("O");
            break;
          case 3:
            Console.Write("@");
            break;
        }
      }
      Console.Write("\n");
    }
}
```

Упражнение 4

Нет, по следующим причинам:

- пропущена точка с запятой, завершающая оператор;
- во второй строке производится попытка обратиться к несуществующему шестому элементу blab;
- во второй строке производится попытка присвоить строку, которая не заключена в двойные кавычки.

Упражнение 5

```
static void Main(string[] args)
{
  Console.WriteLine("Введите строку:");
  string myString = Console.ReadLine();
  string reversedString = "";
  for (int index = myString.Length - 1; index >= 0; index--)
  {
    reversedString += myString[index];
  }
  Console.WriteLine("Строка в обратном порядке: {0}", reversedString);
}
```

Упражнение 6

```
static void Main(string[] args)
{
  Console.WriteLine("Введите строку:");
  string myString = Console.ReadLine();
  myString = myString.Replace("no", "yes");
  Console.WriteLine("Заменено \"no\" на \"yes\": {0}", myString);
}
```

Упражнение 7

```
static void Main(string[] args)
{
  Console.WriteLine("Введите строку:");
  string myString = Console.ReadLine();
  myString = "\"" + myString.Replace(" ", "\" \"") + "\"";
  Console.WriteLine("Вокруг слов добавлены двойные кавычки: {0}", myString);
}
```

Или с использованием `String.Split()`:

```
static void Main(string[] args)
{
  Console.WriteLine("Введите строку:");
  string myString = Console.ReadLine();
  string[] myWords = myString.Split(' ');
  Console.WriteLine("Вокруг слов добавлены двойные кавычки:");
  foreach (string myWord in myWords)
  {
    Console.Write("\"{0}\" ", myWord);
  }
}
```

Глава 6

Упражнение 1

Первая функция имеет тип возврата `bool`, но не возвращает значения `bool`.

Вторая функция имеет аргумент `params`, но этот аргумент не находится в конце списка аргументов.

Упражнение 2

```
static void Main(string[] args)
{
  if (args.Length != 2)
  {
    Console.WriteLine("Требуются два аргумента.");
    return;
  }
  string param1 = args[0];
  int param2 = Convert.ToInt32(args[1]);
  Console.WriteLine("Строковый параметр: {0}", param1);
  Console.WriteLine("Целочисленный параметр: {0}", param2);
}
```

Обратите внимание, что этот ответ содержит код, проверяющий применимость двух аргументов, что не было частью вопроса, но выглядит логичным в данной ситуации.

Упражнение 3

```
class Program
{
  delegate string ReadLineDelegate();
  static void Main(string[] args)
  {
    ReadLineDelegate readLine = new ReadLineDelegate(Console.ReadLine);
    Console.WriteLine("Введите строку:");
    string userInput = readLine();
    Console.WriteLine("Вы ввели: {0}", userInput);
  }
}
```

Упражнение 4

```
struct order
{
  public string itemName;
  public int unitCount;
  public double unitCost;
  public double TotalCost()
  {
    return unitCount * unitCost;
  }
}
```

Упражнение 5

```
struct order
{
  public string itemName;
  public int unitCount;
  public double unitCost;
  public double TotalCost()
  {
    return unitCount * unitCost;
  }
  public string Info()
  {
    return "Информация заказа: " + unitCount.ToString() + " " + itemName +
          " по цене $" + unitCost.ToString() + " за каждый, общая цена $" +
          TotalCost().ToString();
  }
}
```

Глава 7

Упражнение 1

Это утверждение верно только для информации, которая должна быть доступной всем сборкам. Гораздо чаще требуется, чтобы отладочная информация записывалась только при использовании отладочных сборок. В этой ситуации версия `Debug.WriteLine()` более предпочтительна.

Использование версии `Debug.WriteLine()` дает также то преимущество, что не компилируется в окончательные сборки, тем самым сокращая размер результирующего кода.

Упражнение 2

```
static void Main(string[] args)
{
```

```
    for (int i = 1; i < 10000; i++)
    {
      Console.WriteLine("Шаг цикла {0}", i);
      if (i == 5000)
      {
        Console.WriteLine(args[999]);
      }
    }
}
```

В VS точка останова может быть размещена на следующей строке:

```
Console.WriteLine("Loop cycle {0}", i);
```

Свойства точки останова понадобится изменить, чтобы критерием счетчика попаданий было “прервать, когда счетчик попаданий будет равен 5000”.

В VCE точка останова может быть размещена на строке, вызвавшей ошибку, поскольку модифицировать свойства точек останова в VCE, как было описано выше, нельзя.

Упражнение 3

Утверждение не верно. Блок `finally` выполняется всегда. Это может случиться после обработки блока `catch`.

Упражнение 4

```
static void Main(string[] args)
{
  Orientation myDirection;
  for (byte myByte = 2; myByte < 10; myByte++)
  {
    try
    {
      myDirection = checked((Orientation)myByte);
      if ((myDirection < Orientation.North) ||
         (myDirection > Orientation.West))
      {
        throw new ArgumentOutOfRangeException("myByte", myByte,
        "Значение должно находиться между 1 и 4");
      }
    }
    catch (ArgumentOutOfRangeException e)
    {
      // Если этот раздел достигнут, то myByte < 1 или myByte > 4.
      Console.WriteLine(e.Message);
      Console.WriteLine("Присваивание значения по умолчанию, Orientation.North.");
      myDirection = Orientation.North;
    }
    Console.WriteLine("myDirection = {0}", myDirection);
  }
}
```

Обратите внимание, что это вопрос с подвохом. Поскольку перечисление основано на типе `byte`, ему может быть присвоено любое значение `byte`, даже если этому значению не назначено имя в перечислении. В приведенном выше коде при необходимости генерируется собственное исключение.

Глава 8

Упражнение 1

В действительности существуют уровни доступности `public`, `private` и `protected`.

Упражнение 2

Утверждение не верно. Деструктор объекта никогда не должен вызываться вручную; исполняющая среда .NET сделает это во время сборки мусора.

Упражнение 3

Нет, статические методы можно вызывать без каких-либо экземпляров класса.

Упражнение 4

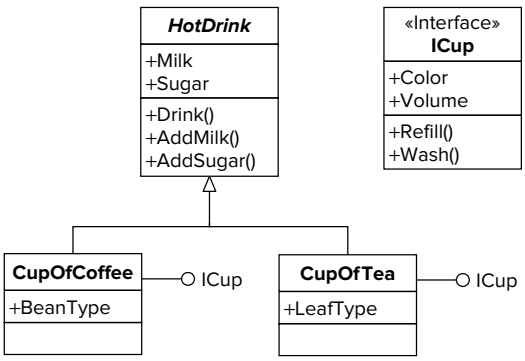


Рис. А.1. *Результирующая UML-диаграмма*

Упражнение 5

```
static void ManipulateDrink(HotDrink drink)
{
  drink.AddMilk();
  drink.Drink();
  ICup cupInterface = (ICup)drink;
  cupInterface.Wash();
}
```

Обратите внимание на явное приведение к `ICup`. Это необходимо, поскольку `HotDrink` не поддерживает интерфейс `ICup`, но известно, что делают два объекта `cup`, которые могут быть переданы этой функции. Однако это опасно, поскольку могут быть переданы и другие классы-наследники `HotDrink`, которые могут и не поддерживать `ICup`. Чтобы исправить это, мы должны проверять, поддерживается ли интерфейс:

```
static void ManipulateDrink(HotDrink drink)
{
  drink.AddMilk();
  drink.Drink();
  if (drink is ICup)
  {
    ICup cupInterface = drink as ICup;
    cupInterface.Wash();
  }
}
```

Используемые здесь операции `is` и `as` описаны в главе 11.

Глава 9

Упражнение 1

Класс `myDerivedClass` наследуется от `MyClass`, но `MyClass` запечатан, поэтому наследовать от него нельзя.

Упражнение 2

Определением их как статических классов или объявлением всех конструкторов приватными.

Упражнение 3

Классы, экземпляры которых невозможно создать, могут быть полезны своими статическими членами. В действительности через эти члены можно даже получить экземпляры таких классов:

```
class CreateMe
{
   private CreateMe()
   {
   }
   static public CreateMe GetCreateMe()
   {
      return new CreateMe();
   }
}
```

Здесь общедоступный конструктор имеет доступ к приватному конструктору, поскольку является частью определения того же класса.

Упражнение 4

Для простоты следующие определения классов показаны как часть одного файла кода, а не в виде листингов отдельных файлов кода для каждого класса:

```
namespace Vehicles
{
   public abstract class Vehicle
   {
   }
   public abstract class Car : Vehicle
   {
   }
   public abstract class Train : Vehicle
   {
   }
   public interface IPassengerCarrier
   {
   }
   public interface IHeavyLoadCarrier
   {
   }
   public class SUV : Car, IPassengerCarrier
   {
   }
   public class Pickup : Car, IPassengerCarrier, IHeavyLoadCarrier
   {
   }
```

```
    public class Compact : Car, IPassengerCarrier
    {
    }
    public class PassengerTrain : Train, IPassengerCarrier
    {
    }
    public class FreightTrain : Train, IHeavyLoadCarrier
    {
    }
    public class T424DoubleBogey : Train, IHeavyLoadCarrier
    {
    }
}
```

Упражнение 5

```
using System;
using Vehicles;
namespace Traffic
{
  class Program
  {
    static void Main(string[] args)
    {
      AddPassenger(new Compact());
      AddPassenger(new SUV());
      AddPassenger(new Pickup());
      AddPassenger(new PassengerTrain());
    }
    static void AddPassenger(IPassengerCarrier Vehicle)
    {
      Console.WriteLine(Vehicle.ToString());
    }
  }
}
```

Глава 10

Упражнение 1

```
class MyClass
{
  protected string myString;
  public string ContainedString
  {
    set
    {
      myString = value;
    }
  }
  public virtual string GetString()
  {
    return myString;
  }
}
```

Упражнение 2

```
class MyDerivedClass : MyClass
{
```

```
    public override string GetString()
    {
      return base.GetString() + " (вывод из производного класса)";
    }
}
```

Упражнение 3

Если метод имеет тип возврата, то его можно использовать как часть выражения:

```
x = Manipulate(y, z);
```

Если никакой реализации частичного метода не предоставлено, он удаляется компилятором наряду со всеми его вызовами. В предыдущем коде это сделало бы результат `x` неясным, поскольку какая-либо замена метода `Manipulate()` не доступна. Возможно, что в отсутствие этого метода просто следовало бы проигнорировать всю строку кода, но компилятор не может решить, это ли вам нужно.

Методы без типа возврата не вызываются как часть выражений, поэтому компилятор может безопасно удалить все ссылки на вызовы частичных методов.

Аналогично, параметры `out` запрещены, поскольку такие переменные должны быть неопределенными перед вызовом метода и становятся определенными после этого вызова. Удаление вызова метода нарушило бы это поведение.

Упражнение 4

```
class MyCopyableClass
{
  protected int myInt;
  public int ContainedInt
  {
    get
    {
      return myInt;
    }
    set
    {
      myInt = value;
    }
  }
  public MyCopyableClass GetCopy()
  {
    return (MyCopyableClass)MemberwiseClone();
  }
}
```

Клиентский код:

```
class Program
{
  static void Main(string[] args)
  {
    MyCopyableClass obj1 = new MyCopyableClass();
    obj1.ContainedInt = 5;
    MyCopyableClass obj2 = obj1.GetCopy();
    obj1.ContainedInt = 9;
    Console.WriteLine(obj2.ContainedInt);
  }
}
```

Этот код выводит `5`, а это показывает, что копируемый объект имеет собственную версию поля `myInt`.

Упражнение 5

```
using System;
using Ch10CardLib;

namespace Exercise_Answers
{
   class Class1
   {
      static void Main(string[] args)
      {
         while(true)
         {
            Deck playDeck = new Deck();
            playDeck.Shuffle();
            bool isFlush = false;

            int flushHandIndex = 0;
            for (int hand = 0; hand < 10; hand++)
            {
               isFlush = true;
               Suit flushSuit = playDeck.GetCard(hand * 5).suit;
               for (int card = 1; card < 5; card++)
               {
                  if (playDeck.GetCard(hand * 5 + card).suit != flushSuit)
                  {
                     isFlush = false;
                  }
               }
               if (isFlush)
               {
                  flushHandIndex = hand * 5;
                  break;
               }
            }

            if (isFlush)
            {
               Console.WriteLine("Флеш!");
               for (int card = 0; card < 5; card++)
               {
                  Console.WriteLine(playDeck.GetCard(flushHandIndex + card));
               }
            }
            else
            {
               Console.WriteLine("Нет флеша.");
            }
            Console.ReadLine();
         }
      }
   }
}
```

В этом коде организован бесконечный цикл, поскольку флеши выпадают нечасто. Может понадобиться нажать <Enter> несколько раз, прежде чем выпадет флеш в перетасованной колоде. Чтобы удостовериться, что все работает должным образом, закомментируйте строку, в которой выполняется тасование колоды.

Глава 11

Упражнение 1

```
using System;
using System.Collections;
namespace Exercise_Answers
{
   public class People : DictionaryBase
   {
      public void Add(Person newPerson)
      {
         Dictionary.Add(newPerson.Name, newPerson);
      }
      public void Remove(string name)
      {
         Dictionary.Remove(name);
      }
      public Person this[string name]
      {
         get
         {
            return (Person)Dictionary[name];
         }
         set
         {
            Dictionary[name] = value;
         }
      }
   }
}
```

Упражнение 2

```
public class Person
{
   private string name;
   private int age;
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }
   public int Age
   {
      get
      {
         return age;
      }
      set
      {
         age = value;
      }
   }
```

```
    public static bool operator >(Person p1, Person p2)
    {
      return p1.Age > p2.Age;
    }
    public static bool operator <(Person p1, Person p2)
    {
      return p1.Age < p2.Age;
    }
    public static bool operator >=(Person p1, Person p2)
    {
      return !(p1 < p2);
    }
    public static bool operator <=(Person p1, Person p2)
    {
      return !(p1 > p2);
    }
}
```

Упражнение 3

```
public Person[] GetOldest()
{
  Person oldestPerson = null;
  People oldestPeople = new People();
  Person currentPerson;
  foreach (DictionaryEntry p in Dictionary)
  {
    currentPerson = p.Value as Person;
    if (oldestPerson == null)
    {
      oldestPerson = currentPerson;
      oldestPeople.Add(oldestPerson);
    }
    else
    {
      if (currentPerson > oldestPerson)
      {
        oldestPeople.Clear();
        oldestPeople.Add(currentPerson);
        oldestPerson = currentPerson;
      }
      else
      {
        if (currentPerson >= oldestPerson)
        {
          oldestPeople.Add(currentPerson);
        }
      }
    }
  }
  Person[] oldestPeopleArray = new Person[oldestPeople.Count];
  int copyIndex = 0;
  foreach (DictionaryEntry p in oldestPeople)
  {
    oldestPeopleArray[copyIndex] = p.Value as Person;
    copyIndex++;
  }
  return oldestPeopleArray;
}
```

Эту функцию усложняет тот факт, что для `Person` операция == не определена, но логика может быть сконструирована и без этого. Вдобавок возврат экземпляра `People` был бы проще, поскольку этим классом легче манипулировать во время обработки. В качестве компромисса экземпляр `People` используется на протяжении всей функции, а в конце преобразуется в массив экземпляров `Person`.

Упражнение 4

```
public class People : DictionaryBase, ICloneable
{
  public object Clone()
  {
    People clonedPeople = new People();
    Person currentPerson, newPerson;
    foreach (DictionaryEntry p in Dictionary)
    {
      currentPerson = p.Value as Person;
      newPerson = new Person();
      newPerson.Name = currentPerson.Name;
      newPerson.Age = currentPerson.Age;
      clonedPeople.Add(newPerson);
    }
    return clonedPeople;
  }
  ...
}
```

Это можно было бы упростить за счет реализации в классе `Person` интерфейса `ICloneable`.

Упражнение 5

```
public IEnumerable Ages
{
  get
  {
    foreach (object person in Dictionary.Values)
      yield return (person as Person).Age;
  }
}
```

Глава 12

Упражнение 1

а), б) и д): да

в) и г): нет, хотя они могут использовать параметры обобщенного типа, предоставленные включающим их классом

е): нет

Упражнение 2

```
public static double? operator *(Vector op1, Vector op2)
{
  try
  {
    double angleDiff = (double)(op2.ThetaRadians.Value -
      op1.ThetaRadians.Value);
    return op1.R.Value * op2.R.Value * Math.Cos(angleDiff);
  }
```

```
    catch
    {
      return null;
    }
  }
```

Упражнение 3

Создавать экземпляр T нельзя, не накладывая на него ограничения new(), которое гарантирует наличие общедоступного конструктора по умолчанию:

```
public class Instantiator<T>
  where T : new()
{
  public T instance;
  public Instantiator()
  {
    instance = new T();
  }
}
```

Упражнение 4

Один и тот же параметр T обобщенного типа используется и в обобщенном классе, и в обобщенном методе. Необходимо переименовать один из них или оба. Например:

```
public class StringGetter<U>
{
  public string GetString<T>(T item)
  {
    return item.ToString();
  }
}
```

Упражнение 5

Ниже показан один из вариантов решения.

```
public class ShortCollection<T> : IList<T>
{
  protected Collection<T> innerCollection;
  protected int maxSize = 10;

  public ShortCollection() : this(10)
  {
  }

  public ShortCollection(int size)
  {
    maxSize = size;
    innerCollection = new Collection<T>();
  }

  public ShortCollection(List<T> list) : this(10, list)
  {
  }

  public ShortCollection(int size, List<T> list)
  {
    maxSize = size;
    if (list.Count <= maxSize)
    {
      innerCollection = new Collection<T>(list);
    }
```

```
    else
    {
      ThrowTooManyItemsException();
    }
  }
  protected void ThrowTooManyItemsException()
  {
    throw new IndexOutOfRangeException(
      "Больше нельзя добавить ни одного элемента, максимальный размер "
      + maxSize.ToString()
      + " элементов.");
  }
  #region IList<T> Members
  public int IndexOf(T item)
  {
    return (innerCollection as IList<T>).IndexOf(item);
  }
  public void Insert(int index, T item)
  {
    if (Count < maxSize)
    {
      (innerCollection as IList<T>).Insert(index, item);
    }
    else
    {
      ThrowTooManyItemsException();
    }
  }
  public void RemoveAt(int index)
  {
    (innerCollection as IList<T>).RemoveAt(index);
  }
  public T this[int index]
  {
    get
    {
      return (innerCollection as IList<T>)[index];
    }
    set
    {
      (innerCollection as IList<T>)[index] = value;
    }
  }
  #endregion
  #region ICollection<T> Members
  public void Add(T item)
  {
    if (Count < maxSize)
    {
      (innerCollection as ICollection<T>).Add(item);
    }
    else
    {
      ThrowTooManyItemsException();
    }
  }
  public void Clear()
  {
    (innerCollection as ICollection<T>).Clear();
  }
```

```
    public bool Contains(T item)
    {
      return (innerCollection as ICollection<T>).Contains(item);
    }
    public void CopyTo(T[] array, int arrayIndex)
    {
      (innerCollection as ICollection<T>).CopyTo(array, arrayIndex);
    }
    public int Count
    {
      get
      {
        return (innerCollection as ICollection<T>).Count;
      }
    }
    public bool IsReadOnly
    {
      get
      {
        return (innerCollection as ICollection<T>).IsReadOnly;
      }
    }
    public bool Remove(T item)
    {
      return (innerCollection as ICollection<T>).Remove(item);
    }
    #endregion
    #region IEnumerable<T> Members
    public IEnumerator<T> GetEnumerator()
    {
      return (innerCollection as IEnumerable<T>).GetEnumerator();
    }
    #endregion
}
```

Упражнение 6

Нет. Параметр типа `T` определен как ковариантный. Но ковариантные параметры типа могут использоваться только как возвращаемые значения метода, а не аргументы метода. Если вы попытаетесь скомпилировать этот код, то получите следующую ошибку компиляции (предполагая, что используется пространство имен `VarianceDemo`):

```
Invalid variance: The type parameter 'T' must be contravariantly valid on
'VarianceDemo.IMethaneProducer<T>.BelchAt(T)'. 'T' is covariant.
```

Неверная вариантность: Параметр типа 'T' должен быть контравариантно верным для 'VarianceDemo.IMethaneProducer<T>.BelchAt(T)'. 'T' является ковариантным.

Глава 13

Упражнение 1

```
public void ProcessEvent(object source, EventArgs e)
{
  if (e is MessageArrivedEventArgs)
  {
    Console.WriteLine("Получено событие Connection.MessageArrived.");
    Console.WriteLine("Message: {0}",
      (e as MessageArrivedEventArgs).Message);
  }
```

```
    if (e is ElapsedEventArgs)
    {
      Console.WriteLine("Получено событие Timer.Elapsed.");
      Console.WriteLine("SignalTime: {0}", (e as ElapsedEventArgs).SignalTime);
    }
  }
  public void ProcessElapsedEvent(object source, ElapsedEventArgs e)
  {
    ProcessEvent(source, e);
  }
```

Обратите внимание на необходимость этого дополнительного метода `ProcessElapsedEvent()`, поскольку делегат `ElapsedEventHandler` не может быть приведен к делегату `EventHandler`. Делать это для делегата `MessageHandler` не понадобится, поскольку он имеет синтаксис, идентичный `EventHandler`:

```
public delegate void MessageHandler(object source, EventArgs e);
```

Упражнение 2

Модифицируйте файл `Player.cs` следующим образом (изменен один метод и добавлено два новых):

```
public bool HasWon()
{
  // Создать временную копию раскладки на руках, которая может изменяться.
  Cards tempHand = (Cards)hand.Clone();
  // Найти наборы по три и четыре карты
  bool fourOfAKind = false;
  bool threeOfAKind = false;
  int fourRank = -1;
  int threeRank = -1;
  int cardsOfRank;
  for (int matchRank = 0; matchRank < 13; matchRank++)
  {
    cardsOfRank = 0;
    foreach (Card c in tempHand)
    {
      if (c.rank == (Rank)matchRank)
      {
        cardsOfRank++;
      }
    }
    if (cardsOfRank == 4)
    {
      // Пометить набор из четырех карт
      fourRank = matchRank;
      fourOfAKind = true;
    }
    if (cardsOfRank == 3)
    {
      // Два набора по три карты означают невозможность выигрыша
      // (threeOfAKind будет true, только если этот код уже выполнялся)
      if (threeOfAKind == true)
      {
        return false;
      }
      // Пометить набор из трех карт
      threeRank = matchRank;
      threeOfAKind = true;
    }
  }
```

```
// Проверить простое условие выигрыша
if (threeOfAKind && fourOfAKind)
{
  return true;
}
// Упростить раскладку на руках, если найдет набор из трех или четырех карт,
// удалив использованные карты
if (fourOfAKind || threeOfAKind)
{
  for (int cardIndex = tempHand.Count - 1; cardIndex >= 0; cardIndex--)
  {
    if ((tempHand[cardIndex].rank == (Rank)fourRank)
      || (tempHand[cardIndex].rank == (Rank)threeRank))
    {
      tempHand.RemoveAt(cardIndex);
    }
  }
}
// В этой точке из метода может быть возврат, поскольку:
// - найден набор из четырех и набор из трех карт, выигрыш
// - найдено два набора из трех карт, проигрыш
// Если возврат из метода не произошел, это означает одну из следующих ситуаций:
// - ни одного набора не найдено, и tempHand содержит 7 карт
// - найден набор из трех карт, и tempHand содержит 4 карты
// - найден набор из четырех карт, и tempHand содержит 3 карты
// Найти исход из четырех наборов, начиная с поиска карт той же самой масти
// показанным ранее способом
bool fourOfASuit = false;
bool threeOfASuit = false;
int fourSuit = -1;
int threeSuit = -1;
int cardsOfSuit;
for (int matchSuit = 0; matchSuit < 4; matchSuit++)
{
  cardsOfSuit = 0;
  foreach (Card c in tempHand)
  {
    if (c.suit == (Suit)matchSuit)
    {
      cardsOfSuit++;
    }
  }
  if (cardsOfSuit == 7)
  {
    // Если все карты имеют ту же самую масть, возможны, но не очевидны два исхода
    threeOfASuit = true;
    threeSuit = matchSuit;
    fourOfASuit = true;
    fourSuit = matchSuit;
  }
  if (cardsOfSuit == 4)
  {
    // Пометить набор из четырех карт одной масти
    fourOfASuit = true;
    fourSuit = matchSuit;
  }
  if (cardsOfSuit == 3)
  {
    // Пометить набор из трех карт одной масти
    threeOfASuit = true;
```

```
        threeSuit = matchSuit;
      }
    }
    if (!(threeOfASuit || fourOfASuit))
    {
      // Чтобы можно было продолжить, нужен, по меньшей мере, один исход
      return false;
    }
    if (tempHand.Count == 7)
    {
      if (!(threeOfASuit && fourOfASuit))
      {
        // Нужны наборы из трех и четырех карт одной масти
        return false;
      }
      // Создать два временных набора для проверки
      Cards set1 = new Cards();
      Cards set2 = new Cards();
      // Если все 7 карт имеют ту же самую масть...
      if (threeSuit == fourSuit)
      {
        // Получить минимальное и максимальное достоинства карт
        int maxVal, minVal;
        GetLimits(tempHand, out maxVal, out minVal);
        for (int cardIndex = tempHand.Count - 1; cardIndex >= 0; cardIndex--)
        {
          if (((int)tempHand[cardIndex].rank < (minVal + 3))
          || ((int)tempHand[cardIndex].rank > (maxVal - 3)))
          {
            // Удалить все карты из набора из трех карт, которые
            // начинаются с minVal или заканчиваются на maxVal
            tempHand.RemoveAt(cardIndex);
          }
        }
        if (tempHand.Count != 1)
        {
          // Если осталось более одной карты, двух исходов не будет
          return false;
        }
        if ((tempHand[0].rank == (Rank)(minVal + 3))
          || (tempHand[0].rank == (Rank)(maxVal - 3)))
        {
          // Если запасная карта может превратить один из наборов из трех карт
          // в набор из четырех карт, значит, есть два набора
          return true;
        }
        else
        {
          // Если запасная карта не подходит, значит, есть два набора из трех
          // карт, но ни одного набора из четырех карт
          return false;
        }
      }
      // Если наборы из трех и четырех карт отличаются...
      foreach (Card card in tempHand)
      {
        // Разделить карты в наборы
        if (card.suit == (Suit)threeSuit)
        {
          set1.Add(card);
        }
```

```
            else
            {
               set2.Add(card);
            }
         }
         // Проверить, последовательны ли наборы
         if (isSequential(set1) && isSequential(set2)) {
            return true;
         }
         else
         {
            return false;
         }
      }
      // Если остается четыре карты (найдено три карты одного достоинства)
      if (tempHand.Count == 4)
      {
         // Если осталось четыре карты, они должны быть той же самой масти
         if (!fourOfASuit)
         {
            return false;
         }
         // Выигрыш, если карты последовательны
         if (isSequential(tempHand))
         {
            return true;
         }
      }
      // Если остается три карты (найдено четыре карты одного достоинства)
      if (tempHand.Count == 3)
      {
         // Если осталось три карты, они должны быть той же самой масти
         if (!threeOfASuit)
         {
            return false;
         }
         // Выигрыш, если карты последовательны
         if (isSequential(tempHand))
         {
            return true;
         }
      }
      // Вернуть false, если не существует двух допустимых наборов
      return false;
   }
   // Служебный метод для получения минимального и максимального достоинств карт
   // (предполагается, что они тоже же самой масти)
   private void GetLimits(Cards cards, out int maxVal, out int minVal)
   {
      maxVal = 0;
      minVal = 14;
      foreach (Card card in cards)
      {
         if ((int)card.rank > maxVal)
         {
            maxVal = (int)card.rank;
         }
         if ((int)card.rank < minVal)
         {
            minVal = (int)card.rank;
```

```
      }
    }
  }
  // Служебный метод для просмотра, находятся ли карты в исходе
  // (предполагается, что они тоже же самой масти)
  private bool isSequential(Cards cards)
  {
    int maxVal, minVal;
    GetLimits(cards, out maxVal, out minVal);
    if ((maxVal - minVal) == (cards.Count - 1))
    {
      return true;
    }
    else
    {
      return false;
    }
  }
```

Глава 14

Упражнение 1

Для того чтобы использовать инициализатор объекта с классом, должен быть включен конструктор по умолчанию без параметров. Можно либо добавить его к этому классу, либо удалить существующий конструктор не по умолчанию. Затем можно использовать следующий код для создания экземпляра и инициализации этого класса за один шаг:

```
Giraffe myPetGiraffe = new Giraffe
{
  NeckLength = "3.14",
  Name = "Gerald"
};
```

Упражнение 2

Не верно. Когда вы используете ключевое слово `var` для объявления переменной, такая переменная все равно будет строго типизированной, просто ее тип определяется компилятором.

Упражнение 3

Можно использовать автоматически реализованный метод `Equals()`. Обратите внимание на невозможность применения для этого операции ==, потому что она сравнивает переменные на предмет ссылки на один и тот же объект.

Упражнение 4

Расширяющий метод должен быть статическим:

```
public static string ToAcronym(this string inputString)
```

Упражнение 5

Метод расширения должен быть включен в статический класс, который доступен из пространства имен, содержащего клиентский код. Для этого нужно либо включить код в то же пространство имен, либо импортировать пространство имен, содержащее этот класс.

Упражнение 6

Один из способов сделать это показан ниже:

```
public static string ToAcronym(this string inputString)
{
   return inputString.Trim().Split(' ')
      .Aggregate<string, string>("",
         (a, b) => a + (b.Length > 0 ?
         b.ToUpper()[0].ToString() : ""));
}
```

Здесь используется тернарная операция для предотвращения ошибок, вызванных множественными пробелами. Обратите также внимание, что требуется версия `Aggregate()` с двумя параметрами обобщенного типа, поскольку необходимо начальное значение.

Глава 15

Упражнение 1

Файл `Program.cs` в проекте Windows Forms содержит метод `Main()` приложения. По умолчанию этот метод выглядит примерно так:

```
[STAThread]
static void Main()
{
   Application.EnableVisualStyles();
   Application.SetCompatibleTextRenderingDefault(false);
   Application.Run(new Form1());
}
```

Визуальным стилем окон форм управляет следующая строка:

```
Application.EnableVisualStyles();
```

Имейте в виду, что в среде Windows 2000 эта строка ничего не делает.

Упражнение 2

Элемент управления `TabControl` поддерживает событие по имени `SelectedINdexChanged`, которое может применяться для выполнения кода, когда пользователь переходит на другую вкладку.

1. В визуальном конструкторе Windows Forms выберите элемент управления `TabCOntrol` и добавьте к нему две вкладки.
2. Назовите новые вкладки Tab Three и Tab Four.
3. При выбранном элементе `TabControl` добавьте событие `SelectedINdexChanged` и перейдите в окно кода.
4. Введите следующий код:

```
private void tabControl1_SelectedIndexChanged(object sender, EventArgs e)
{
   string message = " Вы изменили текущую вкладку с '" +
   tabControl1.SelectedTab.Text + "' на '" +
   tabControl1.TabPages[mCurrentTabIndex].Text + "'";
   mCurrentTabIndex = tabControl1.SelectedIndex;
   MessageBox.Show(message);
}
```

5. Добавьте в начало класса приватное поле mCurrentTabIndex:

```
partial class Form1 : Form
{
private int mCurrentTabIndex = 0;
```

6. Запустите приложение.

По умолчанию первая вкладка, отображаемая в TabControl, имеет индекс 0. Этот факт учитывается при установке поля mCurrentTabIndex в 0. В методе SelectedINdexChanged строится сообщение для отображения. Для этого с помощью свойства SelectedTab получается свойство Text только что выбранной вкладки, а посредством коллекции TabPages извлекается свойство Text вкладки, указанной в поле mCurrentTabIndex. После построения сообщения поле mCurrentTabIndex изменяется, указывая на новую выбранную вкладку.

Упражнение 3

Создавая класс, унаследованный от ListViewItem, вы создаете нечто такое, что может быть использовано вместо "предназначенного" для этого класса ListViewItem. Это значит, что несмотря на то, что сам ListView не имеет дополнительной информации о классе, можно непосредственно сохранять дополнительную информацию в элементах, отображаемых в ListView.

1. Создайте новый класс FQListViewItem:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
namespace ListView
{
  class FQListViewItem : ListViewItem
  {
    private string mFullyQualifiedPath;
    public string FullyQualifiedPath
    {
      get { return mFullyQualifiedPath; }
      set { mFullyQualifiedPath = value; }
    }
  }
}
```

2. Найдите и замените типы ListViewITem на FQListViewItem в файле Form.cs.
3. Найдите и замените все ссылки на .Tag ссылками на .FullyQualifiedPath. В методе listViewFilesAndFolders_ItemActivate приведите выбранный элемент во второй строке на FQListViewITem, как показано ниже:

```
string filename = ((FQListViewItem)lw.SelectedItems[0]).FullyQualifiedPath;
```

Глава 16

Упражнение 1

Для достижения этого понадобится создать одно новое свойство и два события. Начните с создания свойства (private int maxLength = 32767):

```
public int MaxLength
{
  get { return maxLength; }
  set
  {
```

```
    if (value >= 0 && value <= 32767)
    {
      maxLength = value;
      if (MaxLengthChanged != null)
      MaxLengthChanged(this, new EventArgs());
      textBoxText.MaxLength = value;
    }
  }
}
```

Затем создайте два новых события:

```
public event System.EventHandler MaxLengthChanged;
public event System.EventHandler MaxLengthReached;
```

В визуальном конструкторе форм выберите элемент `TextBox` и добавьте обработчик события `TextChanged` со следующим кодом:

```
private void txtLabelText_TextChanged(object sender, EventArgs e)
{
  if (textBoxText.Text.Length >= maxLength)
  {
    if (MaxLengthReached != null)
       MaxLengthReached(this, new EventArgs());
  }
}
```

Максимальная длина текста в нормальном `TextBox` равна размеру типа `System.Int32`, но по умолчанию принимается 32767 символов, чего обычно более чем достаточно. В свойстве на шаге 2 осуществляется проверка, не отрицательно ли значение и больше ли оно 32767, и если это так, запрос изменения игнорируется. Если оказывается, что значение приемлемо, то свойство `MaxLength` элемента `TextBox` устанавливается и инициируется событие `MaxLengthChanged`.

Обработчик события `txtLabelText_TextChanged` проверяет, равно или больше максимальное количество символов в `TextBox` числу, указанному в `maxLength`, и инициирует событие `MaxLengthReached`, если это так.

Упражнение 2

Начните с выбора трех полей в `StatusBar` и изменения значения Bold на `false` (раскройте для этого свойство `Font`). Измените свойство `Enabled` всех трех полей на `True`, затем дважды щелкните на поле Bold и введите следующий код:

```
private void toolStripStatusLabelBold_Click(object sender, EventArgs e)
{
  boldToolStripButton.Checked = !boldToolStripButton.Checked;
}
```

Дважды щелкните на поле Italic и введите следующий код:

```
private void toolStripStatusLabelItalic_Click(object sender, EventArgs e)
{
  italicToolStripButton.Checked = !italicToolStripButton.Checked;
}
```

Дважды щелкните на поле Underline и введите следующий код:

```
private void toolStripStatusLabelUnderline_Click(object sender, EventArgs e)
{
  underlineToolStripButton.Checked = !underlineToolStripButton.Checked;
}
```

Три обработчика событий переключают свойство Checked кнопок панели инструментов. В результате инициализируется событие CheckedChanged. Эти обработчики отвечают за выполнение всей работы, и вы должны модифицировать их так, чтобы также изменялся текст в строке состояния:

```
private void boldToolStripButton_CheckedChanged(object sender, EventArgs e)
{
   Font oldFont, newFont;
   bool checkState = ((ToolStripButton)sender).Checked;
   oldFont = this.richTextBoxText.SelectionFont;
   if (!checkState)
      newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
   else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);
   richTextBoxText.SelectionFont = newFont;
   richTextBoxText.Focus();
   boldToolStripMenuItem.CheckedChanged -= new
      EventHandler(boldToolStripMenuItem_CheckedChanged);
   boldToolStripMenuItem.Checked = checkState;
   boldToolStripMenuItem.CheckedChanged += new
      EventHandler(boldToolStripMenuItem_CheckedChanged);
   // StatusBar
   if (!checkState)
      toolStripStatusLabelBold.Font = new Font(toolStripStatusLabelBold.Font,
      toolStripStatusLabelBold.Font.Style & ~FontStyle.Bold);
   else
      toolStripStatusLabelBold.Font = new Font(toolStripStatusLabelBold.Font,
      toolStripStatusLabelBold.Font.Style | FontStyle.Bold);
}
private void italicToolStripButton_CheckedChanged(object sender, EventArgs e)
{
   Font oldFont, newFont;
   bool checkState = ((ToolStripButton)sender).Checked;
   oldFont = this.richTextBoxText.SelectionFont;
   if (!checkState)
      newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
   else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);
   richTextBoxText.SelectionFont = newFont;
   richTextBoxText.Focus();
   italicToolStripMenuItem.CheckedChanged -= new
      EventHandler(italicToolStripMenuItem_CheckedChanged);
   italicToolStripMenuItem.Checked = checkState;
   italicToolStripMenuItem.CheckedChanged += new
      EventHandler(italicToolStripMenuItem_CheckedChanged);
   // StatusBar
   if (!checkState)
      toolStripStatusLabelItalic.Font = new
         Font (toolStripStatusLabelItalic. Font,
       toolStripStatusLabelItalic.Font.Style & ~FontStyle.Italic);
   else
      toolStripStatusLabelItalic.Font = new
         Font (toolStripStatusLabelItalic. Font,
       toolStripStatusLabelItalic.Font.Style | FontStyle.Italic);
}
private void UnderlineToolStripButton_CheckedChanged(object sender, EventArgs e)
{
   Font oldFont, newFont;
   bool checkState = ((ToolStripButton)sender).Checked;
   oldFont = this.richTextBoxText.SelectionFont;
```

```
    if (!checkState)
      newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
    else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);
      richTextBoxText.SelectionFont = newFont;
      richTextBoxText.Focus();
        underlineToolStripMenuItem.CheckedChanged -= new
          EventHandler(underlineToolStripMenuItem_CheckedChanged);
        underlineToolStripMenuItem.Checked = checkState;
        underlineToolStripMenuItem.CheckedChanged += new
          EventHandler(underlineToolStripMenuItem_CheckedChanged);
    // StatusBar
    if (!checkState)
      toolStripStatusLabelUnderline.Font = new
        Font(toolStripStatusLabelUnderline .Font,
      toolStripStatusLabelItalic.Font.Style & ~FontStyle.Underline);
    else
      toolStripStatusLabelUnderline.Font = new
        Font(toolStripStatusLabelUnderline.Font,
      toolStripStatusLabelItalic.Font.
        Style | FontStyle.Underline);
}
```

Глава 17

Упражнение 1

Развертывание ClickOnce обладает тем преимуществом, что пользователю, устанавливающему приложение, не требуются права администратора. Приложение может быть автоматически установлено по щелчку на гиперссылке. Кроме того, можно сконфигурировать, чтобы новые версии приложения впоследствии также устанавливались автоматически.

Упражнение 2

Манифест приложения описывает приложение и необходимые привилегии, а манифест развертывания — конфигурацию развертывания, такую как политика обновления.

Упражнение 3

Если во время установки необходимы административные права, то вместо развертывания ClickOnce должна использоваться программа установки Windows.

Упражнение 4

Можно использовать редакторы File System Editor, Registry Editor, File Types Editor, User Interface Editor, Custom Actions Editor и Launch Condition Editor.

Глава 18

Упражнение 1

К мастер-странице может быть добавлен элемент управления `LoginView`, чтобы эта информация была доступна каждой странице содержимого. В следующем фрагменте кода можно заметить `LoggedInTemplate` в `LoginView`, который отображается, когда пользователь зарегистрирован. Шаблон `LoggedInTemplate` содержит метку `Label` и кнопку `LinkButton`. Элемент управления `Label` с идентификатором `InfoLabel` используется для отображения информации о пользователе.

```
<asp:LoginView ID="LoginView1" runat="server">
  <LoggedInTemplate>
    <asp:Label ID="InfoLabel" runat="server" Text="Hello, User">
    </asp:Label><br />
    <asp:LinkButton ID="LinkButton1" runat="server"
                    OnClick="OnLogout">Logout</asp:LinkButton>
  </LoggedInTemplate>
</asp:LoginView>
```

Метка `Label` заполняется в отделенном коде, в обработчике `Page_Load`. Имя пользователя может быть доступно через свойство `Context`. Само имя пользователя возвращает свойство `User.Identity.Name`.

```
protected void Page_Load(object sender, EventArgs e)
{
   Control infoLabel = this.LoginView1.FindControl("InfoLabel");
   if (infoLabel != null)
      (infoLabel as Label).Text = "Welcome, " + Context.User.Identity.Name;
}
```

Упражнение 2

В предшествующем коде элемент `DropDownList` имел фиксированный список определенных элементов для предоставления выбора пользователю. Теперь вместо этого используется источник данных `SqlDataSource`, подключаемый к базе данных `Events`:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
  ConnectionString="<%$
    ConnectionStrings:BegVCSharpEventsConnectionString %>"
  SelectCommand="SELECT [Id], [Title], [Date] FROM [Events]
  ORDER BY [Date]">
</asp:SqlDataSource>
```

Вместе с элементом управления `DropDownList` устанавливается идентификатор `DataSourceId`, который ссылается на `SqlDataSource`, а `DataTextField` ссылается на `Title` из выборки SQL для отображения заголовка события:

```
<asp:DropDownList ID="dropDownListEvents" runat="server"
  DataSourceID="SqlDataSource1" DataTextField="Title"
    DataValueField="Id">
</asp:DropDownList>
```

Упражнение 3

При выборе пункта меню File⇨New Project⇨ASP.NET Web Application (Файл⇨Новый проект⇨Веб-приложение ASP.NET) создается проект, включающий множество предварительно созданных элементов. Вы найдете там мастер-страницу по имени `Site.Master`. Мастер-страница применяет таблицу стилей по имени `Site.css`, которую можно найти в папке `Styles`. Внутри мастер-страницы элемент управления `Menu` служит для навигации по сайту. Файлы `Default.aspx` и `About.aspx` используют мастер-страницу и также доступны для навигации.

В подпапке `Account` имеется несколько файлов, использующих средства аутентификации, такие как `Login.aspx`, `Register.aspx` и `ChangePassword.aspx`.

Разработку можно начать с такого проекта и затем добавлять к нему собственные страницы и необходимую функциональность.

Глава 19

Упражнение 1

Создайте новую веб-службу, выбрав пункт меню File⇨New⇨Project (Файл⇨Создать⇨Проект) и выберите шаблон приложения ASP.NET Empty Web Application (Пустое веб-приложение ASP.NET). Назовите его `CinemaReservation`. Добавьте новую веб-сдужбу, выбрав пункт меню Project⇨Add New Item... (Проект⇨Добавить новый элемент), выберите шаблон Web Service (Веб-служба) и назначьте имя `CinemaReservation.asmx`.

Упражнение 2

Эти классы должны выглядеть примерно так, как показано ниже:

```
public class ReserveSeatRequest
{
   public string Name { get; set; }
   public int Row { get; set; }
   public int Seat { get; set; }
}
public class ReserveSeatResponse
{
   public string ReservationName { get; set; }
   public int Row { get; set; }
   public int Seat { get; set; }
}
```

Упражнение 3

Для запоминания зарезервированных мест должен быть объявлен массив `reservedSeats`:

```
private const int maxRows = 12;
private const int maxSeats = 16;
private bool[,] reservedSeats = new bool[maxRows, maxSeats];
```

Реализация метода веб-службы может выглядеть подобно показанному ниже коду. Если запрашиваемое место свободно, оно резервируется и возвращается из веб-службы. Если место занято, возвращается следующее за ним свободное место.

```
[WebMethod]
public ReserveSeatResponse ReserveSeat(ReserveSeatRequest req)
{
   ReserveSeatResponse resp = new ReserveSeatResponse();
   resp.ReservationName = req.Name;
   object o = HttpContext.Current.Cache["Cinema"];
   if (o == null)
   {
      // Заполнить информацию о местах из базы данных или файла...
      HttpContext.Current.Cache["Cinema"] = reservedSeats;
   }
   else
   {
      reservedSeats = (bool[,])o;
   }
   if (reservedSeats[req.Row, req.Seat] == false)
   {
      reservedSeats[req.Row, req.Seat] = true;
      resp.Row = req.Row;
      resp.Seat = req.Seat;
   }
```

```
    else
    {
      int row;
      int seat;
      GetNextFreeSeat(out row, out seat);
      resp.Row = row;
      resp.Seat = seat;
    }
    return resp;
}
```

Упражнение 4

Создайте новое Windows-приложение и добавьте ссылку на веб-службу. Код для вызова веб-службы показан ниже:

```
private void OnRequestSeat(object sender, EventArgs e)
{
  CinemaService.ReserveSeatRequest req =
    new CinemaService.ReserveSeatRequest();
  req.Name = textName.Text;
  req.Seat = int.Parse(textSeat.Text);
  req.Row = int.Parse(textRow.Text);
  CinemaService.CinemaReservationSoapClient ws =
    new CinemaService.CinemaReservationSoapClient();
  CinemaService.ReserveSeatResponse resp =
  ws.ReserveSeat(req);
  MessageBox.Show(String.Format("Reserved seat {0} {1}",
       resp.Row, resp.Seat));
}
```

Глава 20

Упражнение 1

Скопируйте на веб-сайт все файлы, необходимые для запуска веб-приложения. В Visual Studio 2010 имеется диалоговое окно для двунаправленного копирования. Более новые файлы с целевого сервера копируются локально. Если исходный код не должен копироваться на целевой веб-сервер, публикация позволяет создавать сборки и затем копировать на целевой веб-сервер только эти сборки.

Упражнение 2

Копирование сайта требует, чтобы на целевом сервере уже был создан виртуальный каталог. Программа установки позволяет создать виртуальный каталог внутри IIS во время установки.

Упражнение 3

Возможные варианты: публикация в файловой системе, публикация на сервер с установленным компонентом FrontPage Server Extensions, публикация через FTP, а также публикация в 1-Click. В основном это зависит от используемого варианта хостинга и того, что предлагает поставщик хостинга. Во всех случаях на сервере должен быть создан виртуальный каталог. Публикация в файловой системе требует доступа к файловой системе. Это возможно в случае самостоятельного запуска IIS. При публикации с FrontPage Server Extensions этот компонент должен быть установлен на сервере. При публикации через FTP на сервере должен быть установлен FTP-сервер. Публикация 1-Click требует поддержки со стороны поставщика хостинга.

Упражнение 4

Сначала воспользуйтесь инструментом IIS Manager для создания веб-приложения. Затем с помощью Visual Studio скопируйте файлы веб-службы на сервер.

Глава 21

Упражнение 1

```
System.IO
```

Упражнение 2

Объект `FileStream` используется для записи в файл, когда нужен произвольный доступ к файлам или когда речь не идет о строковых данных.

Упражнение 3

- `Peek()` — получает значение следующего символа из файла, но не перемещает позицию чтения в файле
- `Read()` — получает значение следующего символа из файла и перемещает позицию чтения
- `Read(char[] buffer, int index, int count)` — читает `count` символов в `buffer`, начиная с позиции `buffer[index]`
- `ReadLine()` — получает строку текста
- `ReadToEnd()` — получает весь текст файла

Упражнение 4

```
DeflateStream
```

Упражнение 5

Удостоверьтесь, что не установлен атрибут `Serializable`.

Упражнение 6

- `Changed` — происходит, когда файл модифицируется
- `Created` — происходит, когда файл создается
- `Deleted` — происходит, когда файл удаляется
- `Renamed` — происходит, когда файл переименовывается

Упражнение 7

Добавьте кнопку, которая переключает значение свойства `FileSystemWatcher.EnableRaisingEvents`.

Глава 22

Упражнение 1

1. Дважды щелкните на кнопке Create Node (Создать узел) для перехода к обработчику события.

2. Ниже создания `XmlComment` вставьте следующие три строки:

```
XmlAttribute newPages = document.CreateAttribute("pages");
newPages.Value = "1000";
newBook.Attributes.Append(newPages);
```

Упражнение 2

1. `//elements` — возвращает все узлы документа.
2. `element` — возвращает каждый узел элемента в документе, но без корневого элемента.
3. `element[@Type='Noble Gas']` — возвращает каждый элемент, включающий атрибут с именем `Type`, который имеет значение `Noble Gas`.
4. `//mass` — возвращает все узлы с именем `mass`.
5. `//mass/..` — фрагмент `..` заставляет XPath перейти на один уровень выше от выбранного узла; это означает, что данный запрос выбирает все узлы, включающие узел `mass`.
6. `element/specification[mass='20.1797']` — выбирает элемент спецификации, содержащий узел `mass` со значением `20.1797`.
7. `element/name[text()='Neon']` — для выбора узла, содержимое которого проверяется, можно использовать функцию `text()`. Это выбирает имя узла с текстом `Neon`.

Упражнение 3

Вспомните, что XML может быть действительным, правильно оформленным или недействительным. Всякий раз, когда вы выбираете часть документа XML, то имеете дело с фрагментом целого. Это значит, что существует вероятность, что выбранный XML — фактически недействителен сам по себе. Большинство средств просмотра XML откажутся отображать XML, который не является правильно оформленным, поэтому невозможно отобразить результаты многих запросов непосредственно в стандартном средстве просмотра XML.

Глава 23

Упражнение 1

```
static void Main(string[] args)
{
   string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
      "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
   var queryResults =
      from n in names
      where n.StartsWith("S")
      orderby n descending
      select n;
   Console.WriteLine("Names beginning with S:"); // Имена, начинающиеся на S
   foreach (var item in queryResults) {
     Console.WriteLine(item);
   }
   Console.Write("Program finished, press Enter/Return to continue:");
   Console.ReadLine();
}
```

Упражнение 2

Наборы, меньшие чем 5 000 000, не имеют чисел меньше 1000:

```
static void Main(string[] args)
{
   int[] arraySizes = { 100, 1000, 10000, 100000,
                        1000000, 5000000, 10000000, 50000000 };
   foreach (int i in arraySizes) {
      int[] numbers = generateLotsOfNumbers(i);
      var queryResults = from n in numbers
                         where n < 1000
                         select n;
      Console.WriteLine("размер числового массива = {0}: количество(n < 1000) = {1}",
            numbers.Length, queryResults.Count()
      );
   }
   Console.Write("Программа завершена, для продолжения нажмите Enter:");
   Console.ReadLine();
}
```

Упражнение 3

Существенно не влияет на производительность для n < 1000:

```
static void Main(string[] args)
{
  int[] numbers = generateLotsOfNumbers(12345678);
  var queryResults =
     from n in numbers
     where n < 1000
     orderby n
     select n
    ;
  Console.WriteLine("Числа меньше 1000:");
  foreach (var item in queryResults)
  {
    Console.WriteLine(item);
  }
  Console.Write("Программа завершена, для продолжения нажмите Enter:");
  Console.ReadLine();
}
```

Упражнение 4

Очень большие поднаборы, такие как n > 1000 вместо n < 1000, очень медленны:

```
static void Main(string[] args)
{
  int[] numbers = generateLotsOfNumbers(12345678);
  var queryResults =
     from n in numbers
     where n > 1000
     select n
    ;
  Console.WriteLine("Числа меньше 1000:");
  foreach (var item in queryResults)
  {
    Console.WriteLine(item);
  }
  Console.Write("Программа завершена, для продолжения нажмите Enter:");
  Console.ReadLine();
}
```

Упражнение 5

Выводятся все имена, потому что нет запроса.

```
static void Main(string[] args)
{
   string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
      "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
   var queryResults = names;
   foreach (var item in queryResults) {
      Console.WriteLine(item);
   }
   Console.Write("Программа завершена, для продолжения нажмите Enter:");
   Console.ReadLine();
}
```

Упражнение 6

```
var queryResults =
   from c in customers
   where c.Country == "USA"
   select c
  ;
Console.WriteLine("Customers in USA:");
foreach (Customer c in queryResults)
{
   Console.WriteLine(c);
}
```

Упражнение 7

```
static void Main(string[] args)
{
  string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",
     "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
  // Для результирующего набора вроде этого, который состоит только из строк,
  // доступны только методы Min() и Max() (если не используются лямбда-выражения)
  Console.WriteLine("Min(names) = " + names.Min());
  Console.WriteLine("Max(names) = " + names.Max());
  var queryResults =
     from n in names
     where n.StartsWith("S")
     select n;
  Console.WriteLine("Результат запроса: имена, начинающиеся с S");
  foreach (var item in queryResults)
  {
    Console.WriteLine(item);
  }
  Console.WriteLine("Min(queryResults) = " + queryResults.Min());
  Console.WriteLine("Max(queryResults) = " + queryResults.Max());
  Console.Write("Программа завершена, для продолжения нажмите Enter:");
  Console.ReadLine();
}
```

Глава 24

Упражнение 1

Используйте следующий код:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Xml.Linq;
using System.Text;
namespace BegVCSharp_24_exercise1
{
  class Program
  {
    static void Main(string[] args)
    {
      XDocument xdoc = new XDocument(
       new XElement("employees",
        new XElement("employee",
         new XAttribute("ID", "1001"),
         new XAttribute("FirstName", "Fred"),
         new XAttribute("LastName", "Lancelot"),
         new XElement("Skills",
            new XElement("Language", "C#"),
            new XElement("Math", "Calculus")
            )
        ),
        new XElement("employee",
        new XAttribute("ID", "2002"),
        new XAttribute("FirstName", "Jerry"),
        new XAttribute("LastName", "Garcia"),
        new XElement("Skills",
           new XElement("Language", "French"),
           new XElement("Math", "Business")
           )
        )
       )
      );
      Console.WriteLine(xdoc);
      Console.Write("Программа завершена, для продолжения нажмите Enter:");
      Console.ReadLine();
    }
  }
}
```

Упражнение 2

Используйте код, подобный показанному ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
namespace BegVCSharp_24_exercises
{
  class Program
  {
    static void Main(string[] args)
    {
      string xmlFileName =
          @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
      XDocument customers = XDocument.Load(xmlFileName);
      Console.WriteLine("Старейшие клиенты: компании с заказами от 1996 г.:");
      var queryResults =
          from c in customers.Descendants("customer")
          where c.Descendants("order").Attributes("orderYear")
                                  .Any(a => a.Value == "1996")
          select c.Attribute("Company");
```

```
        foreach (var item in queryResults)
        {
          Console.WriteLine(item);
        }
        Console.Write("Программа завершена, для продолжения нажмите Enter:");
        Console.ReadLine();
      }
    }
}
```

Упражнение 3

Ниже показан код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
namespace BegVCSharp_24_exercises
{
  class Program
  {
    static void Main(string[] args)
    {
      string xmlFileName =
        @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
      XDocument customers = XDocument.Load(xmlFileName);
      Console.WriteLine(
        "Компании с индивидуальными заказами на сумму более $10 000");
      var queryResults =
          from c in customers.Descendants("order")
          where Convert.ToDecimal(c.Attribute("orderTotal").Value) > 10000
          select new { OrderID = c.Attribute("orderID"),
                       Company = c.Parent.Attribute("Company") };
      foreach (var item in queryResults)
      {
        Console.WriteLine(item);
      }
      Console.Write("Программа завершена, для продолжения нажмите Enter:");
      Console.ReadLine();
    }
  }
}
```

Упражнение 4

Используйте следующий код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
using System.Text;
namespace BegVCSharp_24_exercises
{
  class Program
  {
    static void Main(string[] args)
    {
    string xmlFileName =
      @"C:\BegVCSharp\Chapter24\Xml\NorthwindCustomerOrders.xml";
```

```
      XDocument customers = XDocument.Load(xmlFileName);
      Console.WriteLine("Клиенты с наибольшими продажами за все время: "+
        "Компании с заказами на общую сумму свыше $100 000");
      var queryResult =
          from c in customers.Descendants("customer")
          where c.Descendants("order").Attributes("orderTotal")
                      .Sum(o => Convert.ToDecimal(o.Value)) > 100000
          select c.Attribute("Company");
      foreach (var item in queryResult)
      {
        Console.WriteLine(item);
      }
      Console.Write("Программа завершена, для продолжения нажмите Enter:");
      Console.ReadLine();
    }
}
```

Упражнение 5

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace BegVCSharp_24_exercise1
{
  class Program
  {
    static void Main(string[] args)
    {
      NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
      Console.WriteLine("Подробные сведения о товарах");
      var queryResults = from p in northWindEntities.Products
                         select new
                         {
                             ID = p.ProductID,
                             Name = p.ProductName,
                             Price = p.UnitPrice,
                             Discontinued = p.Discontinued
                         };
      foreach (var item in queryResults)
      {
        Console.WriteLine(item);
      }
      Console.WriteLine("Подробные сведения о сотрудниках");
      var queryResults2 = from e in northWindDataContext.Employees
                          select new
                          {
                            ID = e.EmployeeID,
                            Name = e.FirstName+" "+e.LastName,
                            Title = e.Title
                          };
      foreach (var item in queryResults2)
      {
        Console.WriteLine(item);
      }
      Console.WriteLine("Для продолжения нажмите Enter:");
      Console.ReadLine();
    }
  }
}
```

Упражнение 6

Используйте код, подобный показанному ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace BegVCSharp_24_exercise6
{
  class Program
  {
    static void Main(string[] args)
    {
      NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
      Console.WriteLine("Самые продаваемые товары (продано на сумму свыше $50 000)");
      var queryResults =
        from p in northWindEntities.Products
        where p.Order_Details.Sum(od => od.Quantity * od.UnitPrice) > 50000
        orderby p.Order_Details.Sum(od => od.Quantity * od.UnitPrice) descending
        select new
        {
          ID = p.ProductID,
          Name = p.ProductName,
          TotalSales = p.Order_Details.Sum(od => od.Quantity * od.UnitPrice)
        };
      foreach (var item in queryResults)
      {
        Console.WriteLine(item);
      }
      Console.WriteLine("Для продолжения нажмите Enter:");
      Console.ReadLine();
    }
  }
}
```

Упражнение 7

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace BegVCSharp_24_exercise7
{
  class Program
  {
    static void Main(string[] args)
    {
      NORTHWNDEntities northWindEntities = new NORTHWNDEntities();
      var totalResults = from od in northWindEntities.Order_Details
                         from c in northWindEntities.Customers
                         where c.CustomerID == od.Order.CustomerID
                         select new
                         {
                           Product = od.Product.ProductName,
                           Country = c.Country,
                           Sales = od.UnitPrice * od.Quantity
                         };
      var groupResults =
        from c in totalResults
        group c by new { Product = c.Product, Country = c.Country } into cg
```

```
            select new {
               Product = cg.Key.Product,
               Country = cg.Key.Country,
               TotalSales = cg.Sum(c => c.Sales)
            }
         ;
         var orderedResults =
            from cg in groupResults
            orderby cg.Country, cg.TotalSales descending
            select cg
         ;
         foreach (var item in orderedResults)
         {
            Console.WriteLine("{0,-12}{1,-20}{2,12}",
               item.Country, item.Product, item.TotalSales.ToString("C2"));
         }
         Console.WriteLine("Для продолжения нажмите Enter:");
         Console.ReadLine();
      }
   }
}
```

Глава 25

Упражнение 1

Не верно. Большая часть кода остается такой же, но есть маленькие отличия, такие как необходимость использования элементов управления `Page` в браузерных приложениях WPF и элементов `Window` — в настольных Windows-приложениях.

Упражнение 2

Для этого необходимо использовать присоединенное свойство. В XAML присоединенные свойства описываются с использованием синтаксиса атрибутов с полностью квалифицированными именами в форме *<ИмяРодительскогоКласса>.<ИмяАтрибута>*. Ниже приведен пример:

```
<Tree>
  <Branch Tree.LeafCount="3" />
  <Branch Tree.LeafCount="42" />
</Tree>
```

Упражнение 3

Утверждения б) и д) верны. Утверждение а) не верно, потому что свойства .NET являются необязательными. Утверждение в) не верно, потому что нет ограничений на свойства зависимости, которые можно иметь для класса. Утверждение г) не верно, потому что это просто соглашение об именовании, основанное на передовом опыте, а не требование.

Упражнение 4

Вы должны использовать элемент управления `StackPanel`.

Упражнение 5

В соглашении об именовании указано, что имя туннелируемого события совпадает с именем, используемым для ассоциированного с ним пузырькового события, но с добавлением префикса `Preview`.

Упражнение 6

Строго говоря, это вопрос с подвохом, поскольку можно выполнить анимацию любого типа свойства. Однако для анимации свойств, типы которых отличаются от `double`, `Color` или `Point`, должны быть созданы собственные классы временных шкал, так что обычно лучше придерживаться перечисленных типов.

Упражнение 7

Динамические ссылки на ресурсы используются, когда необходимо разрешить изменять их во время выполнения или когда ссылка не известна вплоть до времени выполнения.

Глава 26

Упражнение 1

Все перечисленные.

Упражнение 2

Контракт данных с атрибутами `DataContractAttribute` и `DataMemberAttribute`.

Упражнение 3

Используйте расширение `.svc`.

Упражнение 4

Это один из способов, но обычно легче поместить всю конфигурацию в отдельный конфигурационный файл — `web.config` или `app.config`.

Упражнение 5

```
[ServiceContract]
public interface IMusicPlayer
{
   [OperationContract(IsOneWay=true)]
   void Play();
   [OperationContract(IsOneWay=true)]
   void Stop();
   [OperationContract]
   TrackInformation GetCurrentTrackInformation();
}
```

Понадобится также контракт данных для инкапсуляции сведений о треках; в предыдущем коде это `TrackInformation`.

Глава 27

Упражнение 1

Составное действие состоит из двух частей — самого действия и файла (XAML) визуального конструктора, который определяет компоновку действия на поверхности проектирования. Составные действия обычно наследуются от класса `NativeActivity` и предоставляют коллекцию дочерних действий. Например, действие `Sequence` имеет свойство `Activities`, содержащее коллекцию дочерних действий.

Чтобы запланировать запуск дочерних действий, нужно переопределить метод `Execute()` — можно случайным образом выбирать действие или же запускать их все вместе одновременно. Метод `Execute()` принимает экземпляр класса `NativeActivityContext`, который служит для планирования выполнения дочерних действий.

Последний шаг — это создание визуального конструктора, который позволит пользователю перетаскивать действия в ваше действие. Здесь для определения внешнего вида и поведения действия применяется разметка XAML, и часто лучше обратиться к встроенным действиям, чтобы посмотреть, как они реализованы, и повторно использовать некоторые доступные ресурсы XAML. Загрузив инструмент Reflector (`http://reflector.red-gate.com`), можно применять дополнение — средство просмотра BAML для декомпиляции ресурсов, которые используются встроенными сборками, чтобы увидеть XAML, определяющий специальные составные действия.

Упражнение 2

В WF 4 имеется несколько действий, которые позволяют представить рабочий поток как службу WCF. Простейший способ предусматривает выбор шаблона проекта WCF Workflow Service Application (Приложение службы WCF с рабочими потоками) из группы WCF в диалоговом окне создания нового проекта. После этого можно добавлять действия для обработки входящих обращений к методам и при необходимости возврата результатов вызывающему коду.

Упражнение 3

В WF реализована концепция службы Persistence (Постоянство), которая позволяет сохранять и перезагружать экземпляр рабочего потока. Когда рабочий поток не активен (т.е. ожидает некоторого внешнего ввода либо простаивает), он становится кандидатом на сохранение. Если поставщик постоянства настроен, то рабочий поток будет сохранен в него. В готовом виде доступен поставщик SQL Server (см. класс `SqlWorkflowInstanceStore` из сборки `System.Activities.DurableInstancing`). Сохранить рабочий поток невозможно, если он запущен под `WorkflowInvoker` — рабочий поток должен быть размещен с использованием класса `WorkflowApplication` или `WorkflowServiceHost`.

Предметный указатель

Д

З

И

К

Л

М

Н

О

VISUAL C++ 2010 ПОЛНЫЙ КУРС

Айвор Хортон

www.dialektika.com

По существу, в этой книге рассматриваются две обширные темы: язык программирования C++ и программирование приложений Windows с использованием MFC или .NET Framework. Прежде чем вы сможете разработать полнофункциональное приложение Windows, необходимо приобрести хороший уровень знаний языка C++, поэтому упражнения здесь на первом месте.

В первой части книги поэтапно изложены основные темы программирования на языке C++. Вы изучите синтаксис и использование базового языка C++, а также приобретете уверенность и опыт применения его на практике. Модификацию C++/CLI базового языка C++ вы также изучите на практических примерах.

Кроме того, вы узнаете о мощных инструментальных средствах, предоставляемых стандартной библиотекой шаблонов STL, для базовой версии языка C++ и версии C++/CLI. Одна из глав посвящена библиотеке шаблонов для параллельных вычислений, которая позволяет использовать мощь многоядерных PC для приложений с интенсивными вычислениями.

ISBN 978-5-8459-1698-3 **в продаже**

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 2010 И ПЛАТФОРМА .NET 4

5-е издание

Эндрю Троелсен

www.williamspublishing.com

Версия .NET 4 привнесла множество новых API-интерфейсов в библиотеках базовых классов, а также новых синтаксических конструкций в языке C#.
В этой книге вы найдете полное описание всех нововведений в характерной для автора дружественной к читателю манере. Помимо общих вопросов, подробно рассматривается среда Dynamic Language Runtime (DLR); библиотека Task Parallel Library (TPL, включая PLINQ); технология ADO.NET Entity Framework (а также LINQ to EF); расширенное описание API-интерфейса Windows Presentation Foundation (WPF); улучшенная поддержка взаимодействия с COM.
В книге рассматриваются следующие темы

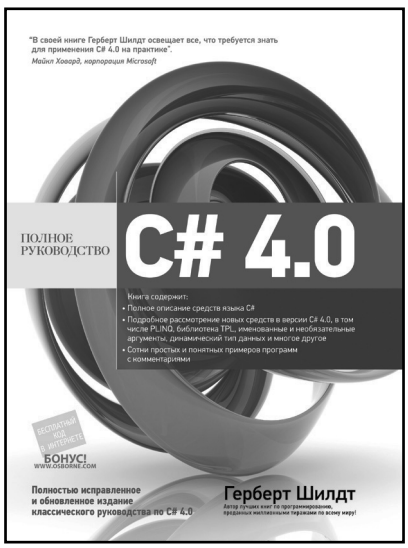
- Особенности платформы .NET 4 и языка Visual C# 2010
- Детали технологии .NET – лидера в производстве современного программного обеспечения
- Полезные советы по разработке от эксперта в .NET, который изучает эту платформу, начиная с ее первой версии
- Полное описание технологий WPF, WCF и WF, поддерживаемых ядром платформы .NET

ISBN 978-5-8459-1682-2

в продаже

C# 4.0
ПОЛНОЕ РУКОВОДСТВО

Герберт Шилдт



www.williamspublishing.com

В этом полном руководстве по C# 4.0 — языку программирования, разработанному специально для среды .NET, — детально рассмотрены все основные средства языка: типы данных, операторы, управляющие операторы, классы, интерфейсы, методы, делегаты, индексаторы, события, указатели, обобщения, коллекции, основные библиотеки классов, средства многопоточного программирования и директивы препроцессора. Подробно описаны новые возможности C#, в том числе PLINQ, библиотека TPL, динамический тип данных, а также именованные и необязательные аргументы. Это справочное пособие снабжено массой полезных советов авторитетного автора и сотнями примеров программ с комментариями, благодаря которым они становятся понятными любому читателю независимо от уровня его подготовки. Книга рассчитана на широкий круг читателей, интересующихся программированием на C#.

ISBN 978-5-8459-1684-6 **в продаже**

C# 4 И ПЛАТФОРМА .NET 4 ДЛЯ ПРОФЕССИОНАЛОВ

Кристиан Нейгел
Билл Ивьен
Джей Глинн
Карли Уотсон
Морган Скиннер

www.dialektika.com

Книга известных специалистов в области разработки приложений с использованием .NET Framework посвящена программированию на языке C# 2010 в среде .NET Framework 4 и в предшествующих версиях. Книгу отличает простой и доступный стиль изложения, изобилие примеров и рекомендаций по написанию высококачественных программ. Подробно рассматриваются такие вопросы, как основы языка программирования C#, организация среды .NET, работа с данными, написание Windows- и веб-приложений, взаимодействие через сеть, создание веб-служб и многое другое. Немалое внимание уделено проблемам безопасности и сопровождения кода. Тщательно подобранный материал позволит без труда разобраться с тонкостями использования Windows Forms и построения веб-страниц. Читатели ознакомятся с работой в Visual Studio 2010, а также с применением различных технологий, встроенных в .NET. Книга рассчитана на программистов разной квалификации.

ISBN 978-5-8459-1656-3

в продаже

SILVERLIGHT 3 С ПРИМЕРАМИ НА C# ДЛЯ ПРОФЕССИОНАЛОВ

Мэтью Мак-Дональд

www.williamspublishing.com

Silverlight 3 — это революционная технология, которая позволяет создавать мощные клиентские приложения, выполняемые в браузерах. Подобно Adobe Flash, Silverlight поддерживает обработку событий, двухмерное рисование, воспроизведение мультимедийного содержимого, сетевые функции, а также анимацию. В то же время технология Silverlight предназначена для платформы .NET и основана на кодах C#.

Наиболее важное преимущество Silverlight заключается в кроссплатформенности. В отличие от обычных приложений .NET, приложения Silverlight могут свободно выполняться в браузерах не от Microsoft (таких, как Firefox) и на других платформах (например, Mac OS X). По сути, Silverlight 3 — это усеченная реализация инфраструктуры .NET, выполняемая в контексте браузера, что делает ее одной из наиболее мощных технологий, представленных компанией Microsoft за последние годы.

ISBN 978-5-8459-1637-2

в продаже