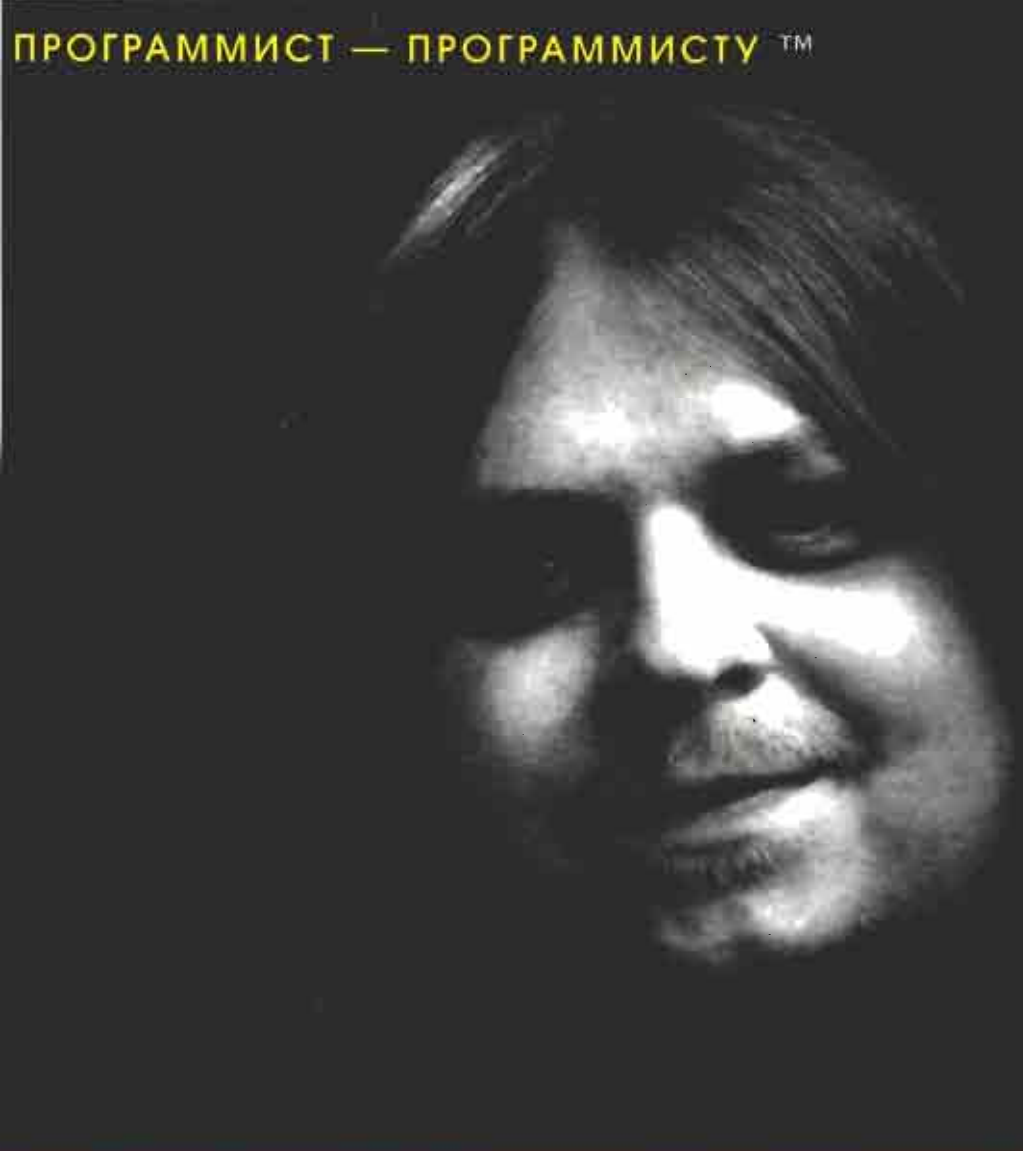
ПРОГРАММИСТ — ПРОГРАММИСТУ ™



C#

Карли Ватсон

совместно с Марко Беллиназо, Олли Корнсом, Дэвидом Эспинозой, Захом Гринфоссом, Кристианом Нейджелом, Джейкобом Хаммером Педерсеном, Джоном Д. Рейдом, Мэттью Рейнольдсом, Морганом Скиннером, Эриком Уайтом

wrox
PROGRAMMER TO PROGRAMMER™

C#

Карли Ватсон
Марко Беллиназо
Олли Корнс
Дэвид Эспиноза
Зах Гринфосс
Кристиан Нейджел
Джейкоб Хаммер Педерсен
Джон Рейд
Мэттью Рейнольдс
Морган Скиннер
Эрик Уайт



Издательство "Лори"

Об авторах

Карли Ватсон

Карли Ватсон — постоянный автор издательства Wrox Press, специалист во многих областях. В начале своей деятельности он намеревался приобрести всемирную известность в области нанотехнологий, поэтому в один прекрасный день его имя может оказаться в списке лауреатов Нобелевской премии. Сегодня интересы Карли в области компьютеров охватывают все, что связано с мобильными устройствами, а также с новейшими технологиями — такими как C#. Он часто пропагандирует новые технологии на различных конференциях, а потом продолжает это занятие на питейных мероприятиях. Кроме того, Карли является энтузиастом сноубординга и жаждет завести кошку.

Благодарность

Благодарю команду издательства Wrox Press за то, что они помогли мне начать писать, а также за работу с результатами этого процесса. Наконец,— и это наиболее важно — я выражаю благодарность моей жене Донне за то, что она продолжает меня терпеть.

Марко Беллиназо

Марко является независимым разработчиком программного обеспечения; он живет в небольшом итальянском городке неподалеку от Венеции. На протяжении нескольких лет Марко работал с VB, C/C++ и другими инструментами компании Microsoft, специализируясь в области разработки и программирования пользовательского интерфейса, API, ActiveX/COM; в настоящее момент он занимается .NET Framework — как с C#, так и с VB.NET. Марко — член команды VB-2-The-Max (www.vb2themax.com), где пишет статьи и создает коммерческое программное обеспечение. Он также исполняет обязанности редактора в двух ведущих итальянских компьютерных журналах: Computer Programming и Visual Basic Journal. С ним можно связаться по адресу: mbellinaso@vb2themax.com.

Олли Корнс

Олли Корнс работает с Интернетом и платформой компании Microsoft с начала 90-х годов. В 1999 г. он стал одним из основателей интернет-компании по ведению электронного бизнеса, главным техническим специалистом которой и является в настоящее время.

Прежде Олли занимался самыми разнообразными вещами, среди которых программирование, написание технических статей и книг, управление сетями, ведение проектов по разработке и консультирование. Он сотрудничал с такими компаниями, как Demon Internet, Microsoft, Saab, Travelstore и Vodafone. Олли имеет ученую степень в области компьютерных наук и сертификат компании Microsoft.

Свободное от работы время он посвящает книгам о возможностях человека, а также практикуется в области китайского искусства внутреннего боевого духа, медитации и лечения. Олли умеет показывать фокусы, связанные с глотанием огня и ножей, однако до сих пор не научился ездить на одноколесном велосипеде.

Дэвид Эспиноза

Дэвид Эспиноза является старшим программистом и владельцем компании Espinosa Consulting. Он родился в Барселоне (Испания). В раннем возрасте Дэвид переехал в Соединенные Штаты, где поступил в университет Невады и получил ученую степень бакалавра в области политики.

Интересы Дэвида сосредоточены на технологиях и инструментах компании Microsoft. В 1999 г. он сотрудничал с Microsoft в качестве ведущего автора сертификационного экзамена по Desktop Visual FoxPro. В последнее время Дэвид занялся электронной коммерцией и решениями по объединению данных; кроме этого, он сотрудничает с производственной компанией, отпочковавшейся от фирмы Reno (штат Невада).

Посвящение

Я хочу посвятить свою работу маме и папе, которые стольким пожертвовали ради того, чтобы у меня были лучшие возможности.

Кроме того, я хочу посвятить ее трем моим самым любимым девушкам на свете — Синтии, Джейми и Эмили. Спасибо вам за ту поддержку, которую вы мне оказали.

Зах Гринфосс

Зах Гринфосс, обладатель сертификата MCSD, является старшим консультантом в Magenic Technologies — консалтинговой компании, имеющей золотой сертификат Microsoft и расположенной в Северной Калифорнии. Зах специализируется в области архитектуры среднего звена и разработок, в которых используются различные технологии, включая COM+, MSMQ, BizTalk и XML. До сотрудничества с Magenic Technologies он работал в Defense Manpower Data Center в Монтерее (Калифорния), где занимался разработкой клиент-серверных приложений для Министерства обороны. Зах и его жена Аманда любят путешествовать, лазить по пещерам и возиться со своими двумя кошками. С Захом Гринфоссом можно связаться по адресу: zachg@magenic.com.

Благодарность

Я хочу выразить признательность своей жене Аманде за ее терпение, любовь и за понимание того, сколько времени занимает работа и одновременное написание книги. Также я хочу поблагодарить Кея Ригга за его внимание и указания: без тебя я бы не стал тем, кем являюсь на сегодняшний день. Наконец, я горжусь тем, что являюсь выпускником Калифорнийского государственного университета Monterey Bay.

Кристиан Нейджел

Кристиан Нейджел является специалистом по обучению и консультантом в Global Knowledge — крупнейшей независимой компании, специализирующейся на обучении информационным технологиям. Поработав с платформами PDP 11, VMS и Unix, Кристиан имеет свыше 15 лет стажа разработки программного обеспечения. Он обладает глубоким знанием технологий компании Microsoft и является сертифицированным специалистом по обучению (Microsoft Certified Trainer, MCT), разработчиком решений (Solution Developer, MCSD) и системным инженером

(Systems Engineer, MCSE); Кристиан получает удовольствие, обучая других людей программированию и разрабатывая распределенные решения. Являясь основателем группы, которая сейчас называется .NET User Group Austria (Группа австрийских пользователей .NET), и региональным директором MSDN, он выступает в роли докладчика на европейских конференциях (TechEd, VCDC), и к нему обращается все большее и большее количество разработчиков. Связаться с Кристианом можно через его web-сайт по адресу http://christian.nagel.net.

Благодарность

Я хотел бы поблагодарить сотрудников издательства Wrox, с подачи которых я начал писать книги; Герти Неймюллера и Гельмута Мая, которые предоставили мне все возможности для создания системы обучения разработке программного обеспечения в Австрии; а также Элизабет за ее огромную поддержку при написании этой книги.

Джейкоб Хаммер Педерсен

Джейкоб Хаммер Педерсен служит системным разработчиком в компании ICL Invia, которая является членом Fujitsu Group.

Джейкоб начал серьезно заниматься программированием с того момента, когда научился выговаривать слово "basic", и, наверное, не случайно для него это по-прежнему основной язык программирования. Он начал программировать на персональных компьютерах в начале 90-х годов на языке Pascal, однако вскоре переключился на язык C++, который ему интересен до сих пор. В середине 90-х гг. Джейкоб снова поменял объект своего внимания, на этот раз на Visual Basic. Летом 2000 г. он узнал о существовании C#, изучением которого и занимается с тех самых пор.

Джейкоб работает в основном с платформами Microsoft, также имеет большой опыт работы с MS Office, COM, COM+ и Visual Basic.NET.

Является гражданином Дании, работает и живет в датском городе Аархус.

Джон Рейд

Джон — старший специалист по технологии в компании Micro Data Base Systems, Inc. (www.mdbs.com), создатель БД TITANIUM и вспомогательной экспертной системы GURU. В настоящее время основной сферой его деятельности является разработка инструментов для работы с базами данных в среде Microsoft .NET. Джон Рейд — редактор компонентов C++ и Object Query Language (OQL — язык запроса объектов) стандарта Object Data Management Group (ODMG — группа по работе с объектными данными), а также соавтор нескольких книг, вышедших в издательстве Wrox, среди которых *ADO.NET Programmer's Reference* и *Professional SQL Server 2000 XML*. Когда Джон не работает, не пишет и не катается на велосипеде, он проводит время с женой и двумя маленькими сыновьями. Джон благодарит свою семью и сотрудников издательства Wrox (в особенности Адриана, Джулиана, Аврила и Роба) за их поддержку и воодушевление.

Мэттью Рейнольдс

Поработав над несколькими проектами в издательстве Wrox Press, Мэттью стал постоянным автором этого издательства и занимается практически всеми аспектами Microsoft.NET. Он также является постоянным автором таких сайтов

издательства Wrox, как ASPToday ("ASP сегодня"), C#Today ("C# сегодня") и Web Services Architect ("Разработчик web-служб"). Мэттью живет и работает в северном Лондоне, связаться с ним можно по адресу: matthewr@wrox.com.

Благодарность

Посвящается Фанджееву Сарину.

Выражаю большую благодарность всем, кто перечислен ниже, за их поддержку и помощь в написании настоящей книги: Лену, Эдварду, Даррену, Алексу, Джо, Тиму, Клэр, Мартину, Ниаму, Тому, Олли, Амиру, Гретхен, Бену, Брэндану, Дэнису, Робу, Вэгги, Марку, Элен, Джеймсу, Зое, Фэйе и Саре. Также большое спасибо моим новым друзьям из издательства Wrox, среди которых Шарлотта, Лора, Карли, Дом С, Дом Л, Ян, Кэйт, Джой, Пит, Хелен, Вики, Джон, Дэйв, Эдам, Крейг, Джейк, Джулиан и Пол.

Морган Скиннер

Морган начал свою компьютерную карьеру в нежном возрасте — в школе на ZX80, когда его не удовлетворила какая-то программа, составленная учителем, и он посчитал, что может написать лучшую на ассемблере. Прикипев душой к Z80 (который ему нравится гораздо больше, чем 6502), он постепенно перешел от школьных ZX81-х к своему собственному ZX Spectrum.

С тех пор Моргану пришлось иметь дело с самыми разными языками программирования и самыми различными платформами, в том числе с VAX Macro Assembler, Pascal, Modula2, SmallTalk, ассемблером x86, PowerBuilder, C/C++, Visual Basic, PL/SQL, TSQL, а теперь он пишет на C#. Он умудрился проработать в одной компании около 12 лет во многом благодаря разнообразию выполняемых задач и хорошей рабочей среде.

В свободное от работы время Морган тратит большие деньги на велосипеды и "расслабляется", воюя с сорняками на своем огороде.

Связаться с Морганом можно по электронной почте по следующему адресу: morgan.skinner@totalise.co.uk.

Эрик Уайт

Эрик Уайт — независимый консультант, специализирующийся по разработкам в области управления офшорами совместно с несколькими ведущими разработчиками из Индии. Написав много больше миллиона строк кода, Эрик имеет свыше 20 лет опыта в создании информационных систем управления, систем для работы с банковскими счетами и других экономических систем и приложений для работы с базами данных. Он, в частности, интересуется методологией объектно-ориентированных разработок, в том числе анализом вариантов использования, UML и шаблонами для разработок. За долгие годы Эрик успел поработать с таким количеством технологий, что их невозможно здесь перечислить; в настоящее время он специализируется по C#, VB.NET, ASP.NET, ADO.NET, XML, COM+, GDI+, SQL Server, а также другим технологиям компании Microsoft.

Эрик любит встречаться с новыми людьми и путешествовать по удаленным местам: с равной легкостью его можно встретить на улицах Бангалора, Лондона или Сан-Франциско. В те моменты, когда он не сидит перед компьютером, он любит лазать по горам в Колорадо и в Индии. С ним можно связаться по адресу: eric@ericwhite.com.

Содержание

Глава 1 Знакомство с C# 1

Что такое .NET Framework 1
Из чего состоит .NET Framework 2
Как можно создавать приложения с помощью .NET Framework 2
Что такое C# 6
Какие типы приложений могут создаваться на C# 6
C# в этой книге 7
Visual Studio.NET 7
Решения VS 9
Итоги 9

Глава 2 Создание программы на C# 10

Среда разработки Visual Studio.NET 10
Консольные приложения 12
Приложения Windows Forms 17
Итоги 20

Глава 3 Переменные и выражения 21

Базовый синтаксис C# 21
Базовая структура консольного приложения на C# 24
Переменные 25
Простые типы 25
Именование переменных 30
Литеральные значения 32
Объявление переменных и присваивание им значений 34
Выражения 35
Математические операторы 35
Операторы присваивания 40
Старшинство операторов 41
Пространства имен 41
Итоги 44
Упражнения 45

Глава 4 Управление порядком выполнения 47

Булева логика 47
Операторы работы с битами 50
Логические операторы присваивания 53
Старшинство операторов с дополнениями 55
Оператор goto 56
Ветвление 57
Тринарный оператор 57
Оператор if 58
Оператор switch 61
Организация циклов 64

Цикл do 65
Цикл while 67
Цикл for 69
Прекращение выполнения цикла 74
Бесконечные циклы 75
Итоги 75
Упражнения 76

Глава 5 Дополнительные сведения о переменных 77

Преобразование типов 77
Сложные типы переменных 85
Перечислимый тип 85
Структуры 90
Массивы 92
Действия над строками 99
Итоги 103
Упражнения 104

Глава 6 Функции 105

Описание и использование функций 106
Возвращаемые значения 108
Параметры 110
Область действия переменных 117
Область действия переменных и другие программные конструкции 120
Функция Main() 123
Функции структур 125
Перегрузка функций 126
Делегаты 128
Итоги 131
Упражнения 132

Глава 7 Отладка и обработка ошибок 133

Отладка в Visual Studio 134
Отладка без использования режима останова (в нормальном режиме) 134
Отладка в режиме останова 141
Обработка ошибок 150
Исключительные ситуации 150
Итоги 158
Упражнения 158

Глава 8 Введение в объектно-ориентированное программирование 160

Что такое объектно-ориентированное программирование 161
Что такое объект 162
Объектом может быть все 165
Жизненный цикл объекта 165
Статические члены класса и члены класса экземпляра 167
Методы ООП 167
Интерфейсы 167
Наследование 169
Полиморфизм 170
Взаимоотношения между объектами 172

Перегрузка операторов 173
События 174
Сравнение ссылочных и значимых типов 175
ООП в приложениях Windows 175
Итоги 178
Упражнения 178

Глава 9 Определение классов 180

Определение классов в C# 180
Определение интерфейсов 183
System.Object 185
Конструкторы и деструкторы 187
Последовательность выполнения конструкторов 188
Инструменты ООП в Visual Studio.NET 191
Окно Class View 191
Окно Object Browser 193
Добавление классов 193
Проекты, представляющие собой библиотеки классов 194
Сравнение интерфейсов и абстрактных классов 197
Типы структур 199
Сравнение неглубокого и глубокого копирования 201
Итоги 202
Упражнения 203

Глава 10 Определение членов классов 204

Определение членов 204
Мастера VS для работы с членами 211
Свойства членов 213
Некоторые дополнительные темы, касающиеся членов класса 213
Сокрытие методов базового класса 214
Вызов переопределенных или скрытых методов базового класса 215
Вложенные определения типов 216
Реализация интерфейсов 217
Реализация интерфейсов в классах 218
Пример приложения 220
Планирование приложения 220
Создание библиотеки классов 221
Клиентское приложение для библиотеки классов 224
Итоги 225
Упражнения 226

Глава 11 Дополнительные сведения о классах 227

Семейства 227
Использование семейств 228
Определение семейств 234
Перегрузка операторов 243
Операторы преобразования типов 248
Усовершенствование CardLib, часть 2 249
Более сложные преобразования типов 253
Упаковка и распаковка 254
Оператор is 255
Оператор as 258

Глубокое копирование 259
Усовершенствование CardLib, часть 3 261
Исключительные ситуации, определяемые программистом 263
Усовершенствование CardLib, часть 4 263
Итоги 265
Упражнения 265

Глава 12 События 267

Что такое событие? 267
Использование событий 268
Обработка событий 268
Определение событий 271
Расширение и использование CardLib 278
Клиентская программа карточной игры для CardLib 279
Итоги 286
Упражнения 286

Глава 13 Использование управляющих элементов Windows Form 287

Windows Form Designer 288
Панель инструментов 289
Управляющие элементы 290
Свойства 290
Свойства Anchor и Dock 291
События 292
Управляющий элемент Button 293
Свойства управляющего элемента Button 294
События управляющего элемента Button 294
Управляющие элементы Label и LinkLabel 297
Управляющий элемент Textbox 298
Свойства управляющего элемента Textbox 299
События управляющего элемента Textbox 300
Управляющие элементы RadioButton и CheckBox 309
Свойства управляющего элемента RadioButton 309
События управляющего элемента RadioButton 310
Свойства управляющего элемента CheckBox 310
События управляющего элемента CheckBox 311
Управляющий элемент GroupBox 311
Управляющий элемент RichTextbox 314
Свойства управляющего элемента RichTextbox 315
События управляющего элемента RichTextbox 316
Управляющие элементы ListBox и CheckedListBox 322
Свойства управляющего элемента ListBox 322
Методы управляющего элемента ListBox 323
События управляющего элемента ListBox 324
Управляющий элемент комбинированного окна (ComboBox) 326
Свойства управляющего элемента ComboBox 326
События управляющего элемента ComboBox 327
Управляющий элемент ListView 331
Свойства управляющего элемента ListView 332
Методы управляющего элемента ListView 334
События управляющего элемента ListView 335
ListViewItem 335
ColumnHeader 335
Управляющий элемент ImageList 335

Управляющий элемент StatusBar 342
Свойства управляющего элемента StatusBar 343
События управляющего элемента StatusBar 343
Класс StatusBarPanel 343
Управляющий элемент TabControl 345
Свойства элемента TabControl 346
Итоги 348

Глава 14 Дополнительные возможности Windows Form 349

Меню 349
Использование управляющего элемента Menu 349
Средство управления MenuItem 351
События управляющего элемента MenuItem 352
Средство управления ContextMenu 352
Панели инструментов 356
Свойства управляющего элемента Toolbar 357
События управляющего элемента Toolbar 358
Свойства класса ToolbarButton 358
SDI- и MDI-приложения 363
Создание MDI-приложений 365
Создание управляющих элементов 373
Отладка пользовательских управляющих элементов 379
Расширение управляющего элемента LabelTextbox 381
Итоги 384
Упражнение 385

Глава 15 Использование диалоговых окон 386

Диалоговые окна 386
Испольэование диалоговых окон 388
Диалоговые окна для работы с файлами 389
OpenFileDialog 389
Диалоговое окно SaveFileDialog 400
Вывод на печать 405
Архитектура вывода на печать 406
Вывод на печать нескольких страниц 411
Задание параметров страницы 413
Диалог вывода на печать 416
Предварительный просмотр 419
Диалоговое окно PrintPreviewDialog 419
Класс PrintPreviewControl 420
Диалоговые окна FontDialog и ColorDialog 420
Диалоговое окно FontDialog 420
Диалоговое окно ColorDialog 423
Итоги 424
Упражнения 424

Глава 16 Введение в GDI+ 426

Обзор вывода графической информации 426
Класс Graphics 427
Удаление объектов 428
Система координат 429
Цвета 434

Вывод линий с помощью класса Pen 435
Рисование фигур с помощью класса Brush 437
Графический вывод текста с помощью класс Font 439
Рисование с использованием образов 443
Рисование с помощью класса TextureBrush 445
Рисование образов с помощью класса Pen 447
Графический вывод текста с помощью класса Image 447
Двойная буферизация 448
Дополнительные возможности GDI+ 450
Вырезание отдельных частей рисунков 450
Вывод на печать 451
Предварительный просмотр 451
Пространство имен System.Drawing.Drawing2D 452
Пространство имен System.Drawing.Imaging 452
Итоги 453

Глава 17 Установка приложений Windows 454

Что такое установка? 455
Типы проектов по установке 456
Архитектура Windows Installer компании Microsoft 457
Термины Windows Installer 457
Преимущества Windows Installer 459
Создание пакета инсталляции для приложения Simple Editor 459
Планирование инсталляции 459
Создание проекта 461
Свойства проекта 461
Редакторы начальных установок 464
Редактор файловой системы 464
Редактор типов файлов 469
Редактор условий запуска 470
Редактор пользовательского интерфейса 472
Построение проекта 475
Инсталляция 476
Диалог Welcome 476
Диалог Read Me 476
Диалог License Agreement 477
Диалог Optional Files 477
Диалог Select Installation Folder 477
Диалог Confirm Installation 478
Диалог Progress 478
Диалог Installation Complete 478
Запуск приложения 478
Самовосстановление 479
Обратная инсталляция 479
Итоги 480

Глава 18 Доступ к данным пользователя 481

Доступ к данным в VS 481
Краткая история обеспечения доступа к данным 482
Введение в ADO.NET 483
Просмотр данных в VS 485
Установка соединения с базой данных 485
Таблицы и отношения базы данных 487

Осуществление доступа к базе данных из приложения 489
Мастер настройки конфигурации адаптера данных 489
Вывод полной информации 494
Перемещение по объекту DataSet 495
Добавление списков 496
Добавление табличной сетки 498
Форматирование табличной сетки 498
Обновление базы данных 499
XML 500
XML-документы 500
Пространства имен XML 504
Правильно оформленный и допустимый XML 505
Проверка допустимости XML-документов 505
Автодокументирование на XML в C# 511
Итоги 514
Упражнения 515

Глава 19 Доступ к данным посредством ADO.NET 516

Что такое ADO.NET? 516
Откуда взялось название ADO.NET? 516
Цели, преследовавшиеся при создании ADO.NET 517
Обзор классов и объектов ADO.NET 518
Объекты-провайдеры 519
Объекты-потребители 520
Использование пространства имен System.Data 520
Осуществление доступа к нескольким таблицам объекта DataSet 540
XML и ADO.NET 547
Поддержка XML в объектах DataSet ADO.NET 547
Поддержка SQL в ADO.NET 548
Использование SQL-команд в объектах DataAdapter 549
Непосредственное выполнение SQL-команд 551
Дополнительная информация об ADO.NET 553
Итоги 554
Упражнения 554

Глава 20 Работа с файлами 556

Потоки 556
Классы для ввода и вывода 557
Классы File и Directory 558
Класс FileInfo 559
Класс DirectoryInfo 560
Объект FileStream 560
Объект StreamWriter 567
Объект StreamReader 569
Дополнительные темы 574
Наблюдение за состоянием файловой структуры 574
Асинхроный доступ к файлам 580
Итоги 585
Упражнения 586

Глава 21 Модули .NET 587

Компоненты 587
Преимущества компонентов 588
Краткая история компонентов 588
Самоописание 589
Модули .NET и библиотека классов .NET Framework 590
Одновременное использование разных языков программирования 590
Самоописывающиеся модули для COM и другого унаследованного кода 591
Структура модулей 592
Создание модулей 593
Создание библиотеки классов 593
Исходный код для Shapes 593
Компиляция библиотеки классов 594
Просмотр содержимого модуля 595
Включение ILDASM в качестве внешнего инструмента в VS 595
Использование ILDASM 595
Манифесты 597
Атрибуты модулей 599
Файл AssemblyInfo.cs 599
Номера версий 601
Атрибуты версий 601
Атрибут AssemblyVersion 602
Совместимость версий 603
Вызов модулей 604
Компиляция файла shapeuser с помощью командной строки 604
Компиляция файла shapeuser с помощью VS 604
Исследование проекта с помощью ILDASM 605
Частные и общие модули 606
Частные модули 606
Общие модули 606
Поиск модулей 610
Итоги 610
Упражнения 611

Глава 22 Атрибуты 613

Что такое атрибут? 613
Отражение 617
Встроенные атрибуты 620
Нестандартные атрибуты 630
Атрибут TestCaseAttribute 631
Атрибут System.AttributeUsageAttribute 636
Итоги 655

Глава 23 Основы web-программирования 656

Краткая история Интернета 656
World Wide Web 657
Каким образом работает Паутина 657
Протокол HTTP — HyperText Transfer Protocol 659
Web-программирование 101 661
HTML 661
HTML-формы 665
Неизменяемые web-страницы в сопоставлении с активными 669
Эволюция Паутины 675

Web-программирование в .NET 676
ASP.NET 676
WebForms 677
Итоги 678

Глава 24 Приложения ASP.NET 679

Создание web-регистратора 681
Основы создания приложения 681
Добавление активных элементов 683
Вхождения web-регистратора 688
Хранение вхождений web-регистратора 689
Вывод вхождений web-регистратора 691
Вывод времени 696
Улучшение внешнего вида 698
Вывод списков вхождений 702
Создание новых вхождений web-регистратора 709
Редактирование вхождений web-регистратора 714
Сортировка вхождений 718
Использование сеансов и куки 720
Ограничение доступа к web-сайту 720
Итоги 724
Упражнения 724

Глава 25 Web-службы 725

До того как появились web-службы 726
Вызов удаленной процедуры 726
SOAP 727
Где находят применение web-службы 728
Сценарий приложения для организации путешествий 729
Сценарий приложения дистрибьютера книжной продукции 729
Типы клиентских приложений 730
Архитектура приложения 730
Архитектура web-служб 731
Программа поиска для web-служб 731
Обнаружение web-служб 732
Какие методы можно вызвать? 733
Вызов метода 734
Web-службы и .NET Framework 736
Создание web-службы 736
Клиент 737
Создание простой web-службы ASP.NET 738
Автоматически создаваемые файлы 739
Добавление web-метода 740
Тестирование web-службы 741
Реализация программы-клиента 742
Реализация клиента ASP.NET 745
Итоги 746

Практический пример 1 — web-сайт для проведения опросов 747

Web-сайт для проведения опросов 747
Версия 1 748
Требования 748
Разработка объектной модели (Business Logig Tier) 748

Тестирование 755
Структуры данных (уровень данных) 756
Реализация объекта 757
Пользовательский интерфейс (уровень представления) 765
Оценка версии 1 772
Версия 2 774
Дополнительные требования 775
Инструмент для просмотра результатов опроса 775
Один человек — один голос 787
Управление опросами 790
Трехзвенность и приложение Poll 793
Итоги 793

Практический пример 2 — Менеджер рассылки новостей в режиме онлайн 795

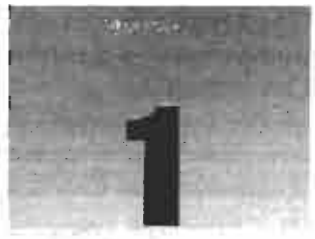
Программа рассылки новостей 795
Общий обзор приложения 796
Модель приложения 797
Объекты данных и бизнес-объекты 798
Модель базы данных 798
Бизнес-классы, предназначенные для доступа в базе данных 801
Управление конфигурацией данных 805
Управление пользовательским интерфейсом 809
Создание пользовательских управляющих элементов 810
Страница управления списками 812
Управление подписчиками 822
Отправка электронной почты подписчикам 826
Класс Utilities 830
Работа с архивом новостей 830
Изменение установок в режиме онлайн 832
Изменение внешнего представления в режиме онлайн 834
Создание подписной формы для пользователя 836
Подписка на список рассылки 837
Обеспечение безопасности приложения 841
Компиляция модуля 845
Запуск приложения 845
Проблемы совместимости браузеров 845
Итоги 846

Приложение A Задание переменной среды PATH 847

Windows 2000/XP 847
Windows NT 848
Windows 95/98/ME 848

Приложение B Инсталляция MSDE 850

Приложение C Дополнительные возможности для компиляции C# 857



Знакомство с C#

Добро пожаловать в первую главу этой книги — "Введение в C#"! Здесь рассматриваются основополагающие вопросы, необходимые для начала работы на C#. В первой главе дается общий обзор C# и .NET Framework, рассказывается, что представляют собой эти технологии, какие причины побуждают ими пользоваться и каким образом эти технологии соотносятся между собой.

Начнем мы с самого общего рассмотрения .NET Framework. Это новая технология, и она включает множество понятий, которые на первый взгляд кажутся хитроумными (так как .NET Framework использует принципиально новый подход к разработке приложений). Отсюда следует, что при обсуждении этой темы вам за короткий промежуток времени придется столкнуться с большим количеством новых понятий. Однако это неизбежно, поскольку знание основ крайне важно для понимания программирования на C#. В последующих главах многие из этих тем будут рассмотрены повторно и более детально.

Далее приводится упрощенное описание собственно языка C#; среди прочего будут рассмотрены его корни и черты сходства с C++.

Наконец, вы познакомитесь с основным инструментом, который будет использоваться на протяжении всей книги,— Visual Studio.NET (VS).

Что такое .NET Framework

.NET Framework — это новая и революционная платформа, созданная компанией Microsoft для разработки приложений.

Самым интересным в этом утверждении является его неопределенность, однако для этого имеются достаточно веские основания. Для начала обратите внимание на то, что в нем не говорится: "для разработки приложений в операционной среде Windows". И хотя первая версия .NET Framework работает под управлением операционной системы Windows, а дальнейшие планы компании входит создание версий, которые будут работать с другими операционными системами, такими как FreeBSD, Linux, Macintosh, и даже на устройствах, относящихся к классу персональных цифровых устройств (personal digital assistant, PDA). Один из основных мотивов создания данной технологии — то, что она предназначается для объединения разнородных операционных систем.

Более того, приведенное выше определение .NET Framework не содержит никаких ограничений относительно типов приложений, создание которых она поддерживает. Причина в том, что никаких ограничений не существует: .NET Framework допускает создание Windows-приложений, web-приложений, web-служб и многого-многого другого, что только можно себе представить.

Технология .NET Framework разрабатывалась таким образом, чтобы ее можно было использовать из любого языка программирования. Это справедливо и по отношению к предмету настоящей книги — C#, а также в отношении C++, Visual Basic, JScript и даже более старых языков, таких как COBOL. Для того чтобы добиться такого результата, пришлось создать специальные версии соответствующих языков: Managed C++ (управляемый C++), Visual Basic.NET, JScript.NET и т. д. — с течением времени появляются все новые и новые. Но все эти языки обладают не только доступом к .NET Framework, но и возможностью взаимодействовать друг с другом. Совершенно нормальной является ситуация, когда разработчики используют в C# код, написанный на Visual Basic.NET, и наоборот.

Все это обеспечивает немыслимый доселе уровень гибкости и является одной из причин, по которой использование .NET Framework представляется таким перспективным.

Из чего состоит .NET Framework

.NET Framework состоит прежде всего из огромной библиотеки программ, к которой можно обращаться из различных языков программирования (например из C#) с помощью различных технологий объектно-ориентированного программирования (ООП, object-oriented programming). Эта библиотека разбита на несколько различных модулей таким образом, что имеется возможность использовать ту или иную ее часть в зависимости от требуемых результатов. Так, например, в одном из модулей содержатся "кирпичики", из которых можно построить Windows-приложения, в другом — "кирпичики", необходимые для организации работы в сети, в третьем — все необходимое для развития Всемирной паутины (Web). Некоторые из этих модулей подразделяются на более специфические подмодули: к примеру, в состав модуля для развития Паутины входит подмодуль, предназначенный для создания web-служб.

Основная идея в данном случае заключается в том, что различные операционные системы могут поддерживать некоторые или все из этих модулей (что зависит от характеристик этих операционных систем). Так, PDA потенциально мог бы поддерживать все основные функциональные возможности .NET, однако маловероятно, чтобы ему потребовались какие-либо более хитрые модули.

Часть библиотеки .NET Framework посвящена описанию некоторых базисных **типов**. Тип — это способ представления данных; определение наиболее фундаментальных из них (например 32-разрядного целого со знаком) облегчает совместное использование языков программирования с помощью .NET Framework. Все вместе это называется **Common Type System (CTS — единая система типов)**.

Кроме вышеописанной библиотеки, в состав системы входит также .NET **Common Language Runtime (CLR — единая система выполнения программ)**, которая ответственна за поддержку выполнения всех приложений, разработанных с использованием библиотеки .NET.

Как можно создавать приложения с помощью .NET Framework

Создание приложений с помощью .NET Framework означает написание программы (на любом из языков программирования, поддерживаемых системой) посредством использования библиотеки программ. В этой книге для наших разработок мы будем использовать VS, который представляет собой мощную интегрированную среду разработки, поддерживающую C# (а также управляемый и не управляемый

C++, Visual Basic.NET и некоторые другие языки программирования). Преимуществом данной среды является та простота, с которой возможности .NET могут быть интегрированы в наши программы. Код, который мы будем создавать, будет целиком написан на C#, однако в нем повсеместно будут использоваться возможности .NET Framework, а там, где возникнет необходимость, мы будем применять в VS дополнительные инструменты.

Для того чтобы написанная на C# программа могла быть выполнена, ее необходимо преобразовать в язык, понятный операционной системе, на которой эта программа будет выполняться; такой язык извсстен под названием родного кода. Подобное преобразование называется **компиляцией** программы и выполняется **компилятором**. Однако в .NET этот процесс состонт из двух этапов.

MSIL и JIT

При компиляции кода, в котором используется библиотека .NET Framework, мы не получаем сразу родной код для конкретной операционной системы. Напротив, наша программа будет откомпилирована в программу на языке **Microsoft Intermediate Language (MSIL — промежуточный язык компании Microsoft)**. Этот код не является специфическим ни для какой операционной системы и так же точно не является специфическим для C#. Компиляция с других языков программирования .NET — например с Visual Basic.NET — на первом этапе тоже происходит на этот язык. Данный этап компиляции выполняется VS в тех случаях, когда мы используем его для разработки приложений на C#.

Очевидно, что для запуска приложения потребуется сделать некоторую дополнительную работу. Эта работа возлагается на **JIT**-компилятор (**Just-In-Time**, своевременный), который осуществляет компиляцию MSIL-кода на код, специфический для ОС и архитектуры используемого компьютера. Только после этого операционная система может выполнить приложение.

В прошлом часто возникала необходимость транслировать написанную программу в несколько различных приложений, каждое из которых предназначалось для конкретной операционной системы и архитектуры центрального процессора. Нередко это был один из способов оптимизации (направленной на то, чтобы программа, к примеру, быстрее выполнялась на AMD-чипах), но иногда это оказывалось критичным (например, когда приходилось использовать некоторые приложения одновременно и в среде Win9x, и в среде WinNt/2000). Теперь такая необходимость исчезла, поскольку JIT-компиляторы (как и предполагает их название) используют код на языке MSIL, который совершенно независим от компьютера, операционной системы и центрального процессора. Существует несколько различных JIT-компиляторов, каждый из которых предназначается для некоторой конкретной архитектуры, поэтому для создания родного кода, необходимого в том или ином случае, будет использоваться соответствующий JIT-компилятор.

Удобство такого подхода заключается в том, что теперь приходится выполнять существенно меньший объем работы: фактически мы получаем возможность просто забыть о системно-зависимых особенностях и сконцентрироваться на более интересных функциональных возможностях создаваемой программы.

Модули

Когда выполняется компилирование приложения, создаваемый MSIL-код сохраняется в некотором **модуле (assembly)**. В состав таких модулей входят как выполняемые файлы приложений, которые могут быть запущены из Windows

напрямую и которым для работы не требуется никаких других программ (такие файлы имеют расширение `.exe`), так и библиотеки, предназначенные для использования другими приложениями (они имеют расширение `.dll`).

Кроме данных на языке MSIL, в модулях хранятся **метаинформация** (т. е. информация об информации, хранящейся в данном модуле, иначе известная под названием **метаданных**) и дополнительные **ресурсы** (дополнительные данные, используемые MSIL, например звуковые и графические файлы). Метаинформация делает модули полностью самоописательными. Для использования модуля не требуется никакой дополнительной информации; другими словами, исключается ситуация, когда не удается добавить необходимые данные в системный регистр, и все подобные ей, что составляло серьезную проблему при разработке приложений на других платформах.

Отсюда следует, что установка приложений зачастую сводится к простому копированию файлов в директорию удаленного компьютера. Поскольку системам, под управлением которых будет выполняться приложение, не требуется никакой дополнительной информации, мы получаем возможность просто запускать выполняемый файл из этой директории (при условии, что на данном компьютере инсталлирована CLR.NET) без каких-либо дополнительных действий.

Естественно, размещать все необходимое для выполнения приложения в одном месте не обязательно. Существует возможность создавать код, выполняющий задачи, которые могут потребоваться нескольким приложениям. В подобных ситуациях полезно располагать такой многократно используемый код в месте, доступном всем приложениям. В .NET Framework таким местом является **Global Assembly Cache (GAC — кэш глобальных модулей)**. Разместить код в таком кэше очень просто: для этого необходимо просто поместить модуль, содержащий соответствующую программу, в директорию, содержащую этот кэш.

Управляемый код

Использование CLR не ограничивается тем моментом, когда мы откомпилировали программу на MSIL, а некоторый JIT-компилятор откомпилировал MSIL в родной код. Код, написанный с помощью .NET Framework, является **управляемым** на этапе выполнения (данный этап зачастую называется **временем выполнения** (runtime)). Это означает, что CLR отслеживает выполнение приложений, управляя памятью, межъязыковой отладкой, обеспечением безопасности и т. п. Напротив, приложения, которые выполняются не под контролем CLR, называются **неуправляемыми**, и в них можно использовать определенные языки программирования, такие как C++, для получения, например доступа к функциям нижнего уровня операционной системы. Однако на C# можно писать только код, который выполняется в управляемой среде. Мы будем использовать возможности CLR по управлению кодом и возложим осуществление любых взаимодействий с операционной системой на .NET.

Сборка мусора

Одной из наиболее важных особенностей управляемого кода является понятие **"сборка мусора"**. Это способ, применяемый в .NET и гарантирующий полное освобождение памяти, использовавшейся приложением, по завершении работы этого приложения. До появления .NET эта задача возлагалась на программистов, и наличие в программе пары простых ошибок могло привести к ситуации, когда огромные блоки памяти таинственным образом исчезали. Обычно это приводило к постепенному замедлению работы компьютера с последующим крахом системы.

Сборка мусора в .NET работает путем как можно более частого инспектирования памяти компьютера и удаления из нее всего, что уже не требуется. Здесь не существует никаких заранее определенных временных рамок; этот процесс может запускаться с частотой несколько тысяч раз в секунду или один раз в несколько секунд, но вы можете быть уверены, что рано или поздно это случиться.

В связи с этим программисты должны учитывать несколько важных аспектов. Поскольку подобные действия выполняются в заранее не известные моменты, приложения нужно разрабатывать с учетом этой особенности. Код, который использует большое количество памяти, должен сам очищать память, а не полагаться на автоматическую сборку мусора, тем более что реализовать это не так уж сложно.

Последовательность действий

Прежде чем двигаться дальше, давайте сведем воедино все обсуждавшиеся шаги, необходимые для создания .NET-приложения:



1. Код приложения записывается на языке программирования, совместимом с .NET, например на C# (см. рис. справа).



2. Полученная программа компилируется на язык MSIL и помещается в модуль (см. рис. слева).

3. Прежде чем этот код начнет выполняться (сам по себе, если он является выполняемым, или при вызове из другой программы), он должен быть откомпилирован в родной код с помощью JIT-компилятора (см. рис. справа).





4. Родной код выполняется в контексте управляемого CLR совместно с другими приложениями или процессами (см. рис. слева).

Связывание

Существует еще один момент, который необходимо учитывать в вышеприведенном процессе. Код на C#, который компилируется на MSIL на шаге 2, совсем не обязательно должен находиться в одном файле. Имеется возможность разнести код приложения по нескольким исходным файлам, которые затем будут скомпилированы в единый модуль. Этот процесс известен под названием **"связывание"** и является чрезвычайно полезным. Причина такого положения дел заключается в том, что работать с несколькими не очень большими файлами намного легче, чем с одним огромным. Можно выделить логически связанный код в отдельный файл, работа над которым будет вестись совершенно независимо и о котором можно забыть после того, как произойдет компиляция. Такой подход существенно упрощает выявление конкретных сегментов кода, когда они оказываются нужными, а заодно позволяет командам разработчиков разделить бремя программирования на отдельные куски, которые затем можно выверять без риска повредить правильно работающие сегменты и те части кода, над которыми трудятся другие программисты.

Что такое C#

Как отмечалось выше, C# — один из языков программирования, который может использоваться для создания приложений, выполняемых в среде .NET CLR. Этот язык — результат эволюции языков C и C++, созданный компанией Microsoft специально для использования на платформе .NET. Являясь новейшей разработкой, C# конструировался очень тщательно, с учетом наилучших возможностей других языков, предназначенных для решения специфических проблем.

Создавать приложения на C# легче, чем на C++, поскольку синтаксис языка C# более простой, чем синтаксис C++. Тем не менее C# является мощным языком программирования, и имеется мало вещей, которые можно сделать на C++ и нельзя на C#. Однако необходимо помнить, что возможности C#, соответствующие наиболее продвинутым возможностям C++ (например непосредственный доступ и внесение изменений в системную память), могут быть осуществлены только с помощью кода, помеченного как **unsafe (небезопасный)**. Эти продвинутые способы программирования, как следует уже из их названия, представляют потенциальную опасность, поскольку они позволяют записывать информацию в критические системные участки памяти с потенциально катастрофическими результатами. Именно по этой причине, а также из некоторых других соображений мы не будем рассматривать данную тему в этой книге.

Иногда код на C# оказывается более объемным, чем на C++. Это следствие того, что в C# (в отличие от C++) осуществляется **контроль безопасности использования типов**. С точки зрения непрофессионала это означает, что если некоторые данные отнесены к определенному типу, то впоследствии они не могут самостоятельно преобразовываться в другие типы. Поэтому, существуют строгие правила, которым необходимо следовать при выполнении преобразования из одного типа в другой. Это часто приводит к необходимости писать на C# больший объем кода, чем на C++. Однако взамен мы получаем то преимущество, что программа становится более надежной и ее отладка упрощается, поскольку .NET постоянно следит за тем, к какому типу относится данная переменная в каждый конкретный момент времени. К примеру, в C# мы не можем дать задание: "Возьми 4 байта памяти из этих данных и 10 байтов из других данных и рассматривай эту область памяти как X", но это не обязательно является недостатком.

C# является всего лишь одним из языков, позволяющих работать в .NET, но на взгляд автора он безусловно лучший. Его главное преимущество заключается в том, что C# — единственный язык, разработанный для .NET Framework "с нуля", и он может использоваться в качестве основного языка для различных версий .NET, которые предназначаются для переноса на другие операционные системы. Для того чтобы некоторые языки — вроде VB.NET — в максимальной степени соответствовали своим предшественникам и при этом могли работать с CLR, приходится отказываться от поддержки отдельных возможностей библиотеки программ .NET. Напротив, C# позволяет использовать любые возможности, предоставляемые библиотекой программ .NET Framework.

Какие типы приложений могут создаваться на C#

Как уже говорилось, .NET Framework не налагает никаких ограничений на возможные типы создаваемых приложений. C# использует эту систему и также не имеет никаких ограничений, касающихся типов приложений. Тем не менее давайте рассмотрим некоторые наиболее часто встречающиеся типы приложений:

- **Приложения Windows.** Это приложения вроде Microsoft Office, имеющие знакомый "Windows-подобный" вид и представление. Создавать такие приложения достаточно просто с помощью модуля .NET Framework, который называется **Windows Forms** и представляет собой библиотеку **управляющих элементов** (кнопок, панелей инструментов, меню и т. п.); эта библиотека может использоваться для создания пользовательского интерфейса (user interface, UI) Windows.
- **Web-приложения.** Эти приложения представляют собой web-страницы, которые могут просматриваться любым web-браузером. В состав .NET Framework входит мощная система динамического создания содержимого web-страниц, позволяющая идентифицировать пользователя, обеспечивать безопасность и пр. Эта система называется **Active Server Pages.NET (ASP.NET — активные серверные страницы .NET)**; для создания приложений ASP.NET можно применять **Web Forms** языка C#.
- **Web-службы.** Это новый замечательный способ создания гибких распределенных приложений. С помощью web-служб можно обмениваться по Интернету практически любыми данными с использованием единого простого синтаксиса независимо от того, какой язык программирования применялся при создании web-службы и на какой системы она размещена.

Приложениям всех перечисленных типов может потребоваться доступ к базам данных, что осуществляется с помощью раздела .NET Framework, называемого **Active Data Objects.NET (ADO.NET — активные объекты с данными .NET)**. Также можно использовать и многие другие ресурсы, например, инструменты для создания сетевых компонентов, графического вывода, выполнения сложных математических вычислений и т. д.

C# в этой книге

Второй и третий разделы данной книги посвящены синтаксису и использованию C#, при этом они практически обходят вниманием .NET Framework. Это совершенно необходимо, поскольку без твердых навыков программирования на C# невозможно использовать .NET Framework. Фактически мы начнем с самых азов, а рассмотрение наиболее сложной темы — объектно-ориентированного программирования (ООП, Object Oriented Programming) — отложим до того момента, когда будут изучены основы. Материал будет излагаться начиная с самых простых принципов, исходя из предположения, что у читателя отсутствуют какие-либо знания по программированию.

После этого мы окажемся готовы к созданию приложений тех типов, которые перечислены выше. В четвертом разделе настоящей книги будет описано программирование с использованием Windows Forms; в пятом разделе рассматриваются другие связанные с .NET темы (например доступ к базам данных); в шестом разделе вы познакомитесь с созданием web-приложений и web-служб. В заключение будут даны более сложные практические примеры, где будет применена информация из предшествующих разделов.

Visual Studio.NET

В данной книге для разработки всех программ на C#, начиная от самых простых примеров использования командной строки и заканчивая более сложными типами создаваемых проектов, мы будем применять Visual Studio.NET (VS).

VS не является обязательным для создания приложений на C#, однако эта система существенно упрощает жизнь программиста. Конечно, мы можем (при желании) производить манипуляции над файлами с исходным кодом на C# с помощью обычного текстового редактора (вроде широко распространенного приложения Notepad) и компилировать код в модули с помощью компилятора командной строки, который является составной частью .NET Framework. Но зачем это делать, если имеется такой мощный инструмент, как VS?

Ниже приводится краткий перечень возможностей VS, благодаря которым эта система является наиболее привлекательным средством разработки в .NET:

- VS автоматически выполняет все шаги, необходимые для компиляции исходного кода, и одновременно позволяет управлять всеми используемыми опциями, если мы пожелаем их переопределить.
- Текстовый редактор VS настроен для работы с теми языками, которые поддерживаются VS (включая C#), поэтому он может интеллектуально обнаруживать ошибки и подсказывать в процессе ввода, какой именно код необходим.
- В состав VS входят программы, позволяющие создавать приложения в Windows Forms и Web Forms путем простого перетаскивания мышью элементов пользовательского интерфейса.
- Многие типы проектов, создание которых возможно на C#, могут разрабатываться на основе "каркасного" кода, заранее включаемого в программу. Вместо того чтобы каждый раз начинать с нуля, VS позволяет использовать уже имеющиеся файлы с исходным кодом, что уменьшает временные затраты на создание проекта.
- В состав VS входит несколько вспомогательных программ, которые позволяют автоматизировать выполнение наиболее распространенных задач; причем многие из этих программ могут добавлять необходимый код в уже существующие файлы, так что программисту не придется беспокоиться (а в некоторых случаях и вообще вспоминать) о соблюдении синтаксических правил.
- VS имеет большое количество мощных инструментов, благодаря которым можно просматривать отдельные элементы проекта или осуществлять в них поиск, независимо от того, являются ли эти элементы файлами с кодами на языке C# или представляют собой какие-либо иные ресурсы, например, двоичные графические или звуковые файлы.
- Распространять приложения в VS столь же просто, как и писать их: VS облегчает передачу кода клиентам и позволяет им инсталлировать его без каких-либо проблем.
- VS допускает использование совершенных методов отладки при разработке проектов: например, пошаговое выполнение кода, когда выполняется один оператор за раз, что дает возможность следить за текущим состоянием приложения.

И это далеко не все преимущества VS!

Решения VS

Когда мы используем VS для разработки приложений, то мы делаем это через создание **решений**. В терминах VS решение — это нечто большее, чем просто приложение. Решения состоят из проектов, среди которых могут быть "проекты Windows Forms", "проекты Web Forms" и т. д. Решения могут включать *несколько* проектов, что позволяет группировать логически взаимосвязанные части кода в одном месте, даже если они в действительности компилируются в разные модули, расположенные в различных местах на жестком диске.

Такая возможность оказывается очень полезной, поскольку она позволяет работать над "разделяемым" кодом (который может быть помещен в кэш глобальных модулей) одновременно с приложениями, использующими этот код. Отладка программ оказывается намного легче, когда имеется единая среда разработки, в которой применяется пошаговое выполнение операторов в различных модулях.

Итоги

В данной главе было дано общее описание .NET Framework и рассмотрено, каким образом эта система упрощает разработку эффективных и гибких приложений. Вы познакомились с процессом превращения программ, написанных на таких языках, как C#, в работающие приложения, и узнали, выгоды можно извлечь из использования управляемого кода, запускаемого под управлением единой системы выполнения программ CLR.NET.

Также было рассказано, что представляет собой C#, какое отношение он имеет к .NET Framework, и дано описание инструмента, который будет использоваться для выполнения разработок на C#,— Visual Studio.NET.

В следующей главе мы с помощью VS создадим работающий код на C#; полученные при этом знания в дальнейшем позволят нам сконцентрироваться собственно на языке C# и не уделять излишнего внимания работе VS.

2

Создание программы на C#

Теперь, когда мы выяснили, что из себя представляет C# и каким образом он вписывается в .NET Framework, настало время заняться грязной работой и написать конкретную программу. На протяжении всей книги мы будем использовать Visual Studio.NET (VS), поэтому первое, что необходимо сделать,— это получить некоторое представление о данной среде разработки. VS является очень большим и очень сложным продуктом, способным навести ужас на тех, кто встречается с ним впервые; однако использовать его для создания несложных приложений на удивление просто. Когда в данной главе мы начнем с ним работать, вы увидите, что немного "поиграть" с программами на C# можно и не обладая большими знаниями. В дальнейшем вам встретятся более сложные операции, которые может выполнять VS, однако на данный момент все, что нам потребуется,— это базовые знания о его возможностях.

Познакомившись с VS, мы создадим два простых приложения. При этом для нас будет важно не столько их программное наполнение, сколько то, что они работают правильно и что мы прошли через все процедуры создания приложений.

Первое приложение, которое мы собираемся создать, будет простым консольным **приложением**. Консольными называются такие приложения, в которых не используется многооконная графическая среда; поэтому нам не придется беспокоиться о кнопках, меню, организации взаимодействия с указателем мыши и тому подобных вещах. Это приложение будет запускаться из окна командной строки, и взаимодействие с ним будет организовано весьма простым способом.

Вторым будет **приложение Windows Forms**. Его внешний вид и работа с ним окажутся очень знакомыми для пользователей Windows, а усилий на его создание потребуется (что удивительно) не так уж много. При этом синтаксис необходимого кода окажется более сложным, хотя во многих случаях нам не придется беспокоиться о деталях.

Приложения обоих типов будут использоваться нами на протяжении следующих двух разделов, причем вначале консольному приложению будет уделяться несколько больше внимания. Дополнительная гибкость оконных приложений не является принципиально важной при изучении языка C#, в то время как простота консольных приложений позволит нам сосредоточиться на изучении синтаксиса и не беспокоиться о внешнем виде приложения.

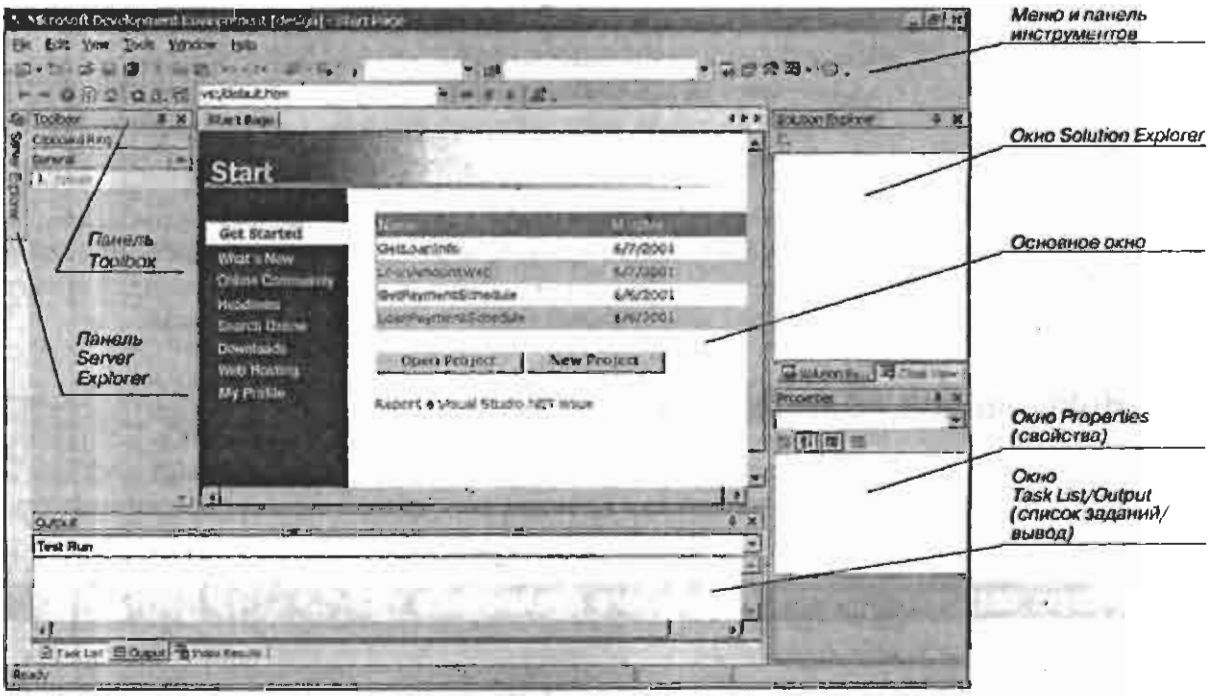
Итак, пора приступать к делу!

Среда разработки Visual Studio.NET

При первой загрузке VS на монитор выводится множество окон, большинство из которых пусты, а также ряд пунктов меню и иконок, которыми нам предстоит пользоваться, но не волнуйтесь заранее: очень скоро они станут более понятными.

Если вы запускаете VS впервые, вам будет предложен список предпочтений, предназначенный для пользователей, которые имеют опыт работы с предыдущими версиями этой среды разработки. На данном этапе установки, используемые по умолчанию, нас вполне устраивают, поэтому все, что от вас требуется,— это просто принять их; в этих установках не содержится ничего такого, чего нельзя было бы изменить впоследствии.

Внешнее представление среды VS является полностью настраиваемым, однако повторим еще раз, что то представление, которое используется по умолчанию, вполне подходит для наших целей. Оно имеет следующий вид:



Основное окно, в котором при запуске VS появляется "начальная страница", в дальнейшем будет содержать весь наш код. Оно поделено на части так, чтобы можно было легко перемещаться между файлами, щелкая мышью по именн соответствующего файла. У этого окна имеются и другие функцин: в нем могут выводиться графические пользовательские интерфейсы, разрабатываемые для наших проектов, файлы с обычным текстом, HTML, а также различные встроенные в VS инструменты. Все эти функцин будут описываться по ходу изложения материала.

Выше основного окна располагаются панели инструментов и меню VS. Здесь могут находиться панели инструментов с самыми разнообразными функциональными возможностями: сохранение и загрузка файлов, создание и запуск проектов, отладка управляющих элементов и т. д. Повторим еще раз, что все эти возможностн будут обсуждаться по мере возникновения необходимости в них.

Ниже приводится краткое опнсание основных возможностей VS, пользоваться которыми приходится наиболее часто:

- Панель Server Explorer и панель Toolbox (инструменты) появляются при движении по ним указателя мыши, открывая различные дополнительные возможности, например, доступ к установкам сервера и службам, а также к элементам построения пользовательских интерфейсов для оконных прнложений.

- В окне Solution Explorer выводится информация о **решении**, загруженном в настоящий момент. "Решение" в терминологии Visual Studio обозначает один или несколько проектов вместе с их настройками. Здесь мы можем увидеть различные представления проектов, участвующих в решении: например, из каких файлов они состоят и что в этих файлах содержится.
- Окно Properties (свойства) позволяет получить более детальное представление о содержимом проекта; в нем предусмотрена возможность дополнительной настройки отдельных элементов. К примеру, здесь можно изменить внешний вид кнопки приложения Windows.
- Окно Task List/Output (список заданий/вывод) содержит информацию, относящуюся к компиляции проектов, а также задачи, которые требуют завершения (они представлены наподобие списка заданий в Microsoft Outlook). Эти задачи могут вводиться вручную, а могут генерироваться VS автоматически.

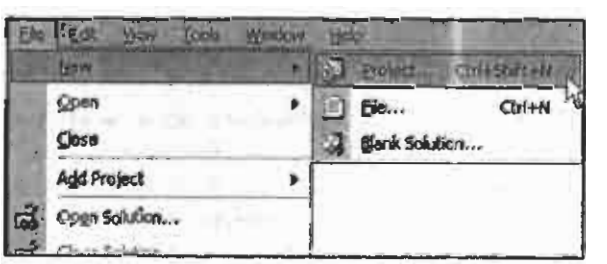
Может показаться, что объем новой информации слишком велик, однако не беспокойтесь: пройдет немного времени, и вы привыкнете к Visual Studio. Итак, давайте приступим к созданию самого первого опытного проекта, который предполагает использование многих элементов VS из числа описанных выше.

Консольные приложения

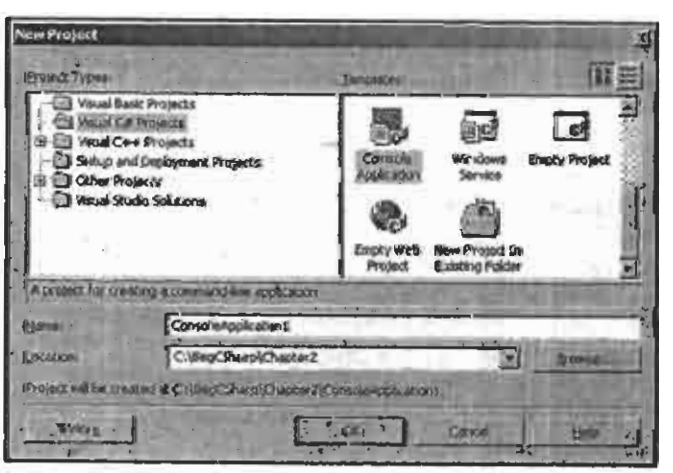
Мы будем использовать консольные приложения на протяжении всей книги, поэтому давайте пройдем шаг за шагом весь процесс создания простого консольного приложения.

Практикум: создание простого консольного приложения

1. Создайте новый проект консольного приложения, выбрав пункт меню File | New | Project... (см. рис. справа).



2. В поле Project Types выберите папку Visual C# Projects, а в поле Templates выберите тип проекта Console Application (для этого вам придется немного прокрутить вниз содержимое поля). Замените текст в поле Location на `C:\BegCSharp\Chapter2` (если этой директории не существует, она будет создана автоматически) и оставьте текст, находящийся в поле Name по умолчанию, без изменений (см. рис. слева).



3. Щелкните по кнопке OK.

4. После того как проект будет инициализирован, добавьте в файл, выведенный в основном окне, следующую строку кода:

```
using System;

namespace ConsoleApplication1
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      static void Main(string[] args)
      {
         //
         // TODO: Add code to start application here
         // static void Main(string[] args)
         Console.WriteLine("The first app in Beginning C# Programming!");
      }
   }
}
```

5. Выберите пункт меню Debug | Start Without Debugging. Через несколько мгновений появится следующее окно:



6. Нажмите любую клавишу для выхода из приложения.

Как это работает

На данном этапе мы не будем анализировать сам код, поскольку пока нас больше интересует использование VS для ввода и запуска программы. Как вы могли заметить, VS делает за нас огромный объем работы, существенно упрощая процесс компиляции и выполнения кода. Однако даже эти простые шаги можно выполнять разными способами. Так, например, создание нового проекта может быть осуществлено с помощью пункта меню File | New | Project... (как было показано выше), посредством нажатия клавиш *Ctrl+Shift+N* либо щелчком мышью по соответствующему значку на панели инструментов.

Наша программа также может быть откомпилирована и выполнена несколькими различными способами. Метод, который мы использовали выше, выбрав пункт меню Debug | Start Without Debugging, имеет два упрощенных способа вызова: с клавиатуры (*Ctrl+F5*) и посредством иконки на панели инструментов. Кроме этого, существует возможность запускать код в режиме отладки с помощью пункта меню Debug | Start (тот же результат можно получить при нажатии клавиши *F5* или при щелчке мышью по соответствующей иконке), или же можно откомпилировать проект, вообще не запуская его (с разрешением отладки или без), воспользовавшись пунктом меню Build | Build, комбинацией клавиш *Ctrl+Shift+B* или еще одной иконкой на панели инструментов. После того как файл откомпилирован,

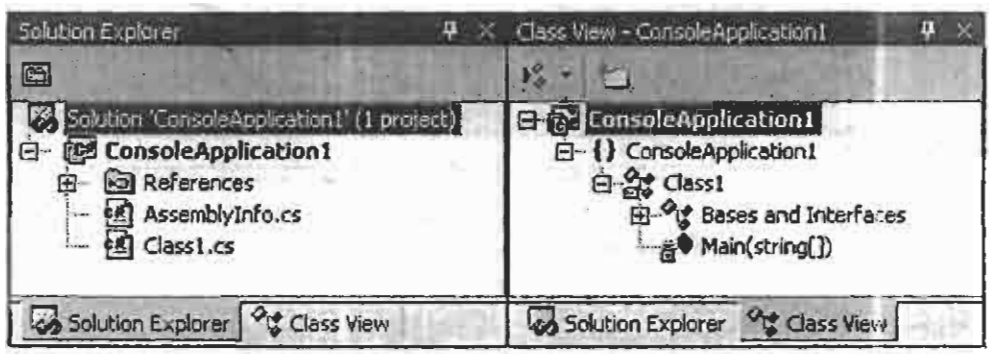
можно выполнить сгенерированный файл с расширением .exe, запустив его либо в Windows Explorer, либо из командной строки. Для этого необходимо открыть окно командной строки, заменить директорию на C:\BegCSharp\Chapter2\ConsoleApplication1\bin\Debug, набрать ConsoleApplication1 и вернуться назад.

В следующих примерах мы будем просто говорить "создайте новый консольный проект" или "выполните код", а вы сможете воспользоваться любым из перечисленных способов. За исключением тех случаев, когда это специально оговаривается, все программы следует выполнять в режиме с разрешенной отладкой.

Имеется один важный момент, на который следует обратить внимание: сообщение **"Press any key to continue"** ("Для продолжения нажмите любую клавишу"), встретившееся нам в этом примере, появляется только в том случае, если код выполняется без разрешения отладки. Если код проекта запускается в отладочном режиме, то консольное окно, в котором он выполняется, закроется тотчас после завершения его работы. Вообще говоря, это правильно, однако в нашем случае такое сообщение было весьма полезным, поскольку окно не закрылось сразу же после своего появления и можно было увидеть результат выполнения приложения.

Теперь, когда мы создали проект, мы можем более детально познакомиться с некоторыми аспектами среды разработки.

Первое окно, которое мы рассмотрим, это окно **Solution Explorer/Class View**, расположенное в правом верхнем углу экрана. Как показано на приведенном ниже рисунке, оно может отображать два режима (переключиться из одного режима в другой можно, щелкнув мышью по одной из кнопок в самом низу окна):

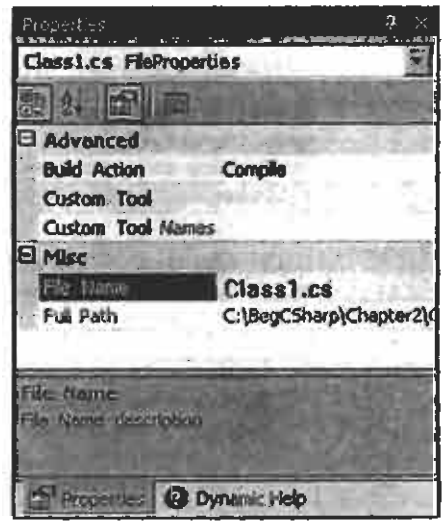


В режиме **Solution Explorer** отображаются файлы, из которых состоит проект ConsoleApplication1. Наряду с файлом Class1.cs, в который мы добавили код, там показан еще один программный файл — AssemblyInfo.cs. (Все файлы, в которых содержатся коды на C#, имеют расширение .cs.) Второй файл с кодом на C# пока не должен нас беспокоить; в нем содержится дополнительная информация о проекте.

Мы можем использовать это окно для внесения изменений в код, который выводится в главном окне; для этого следует либо дважды щелкнуть мышью по нужным .cs-файлам, а затем выбрать в контекстном меню **View Code**, либо сначала выделить их, а затем нажать кнопку панели инструментов. В этом же окне мы можем провести ряд других операций над этими файлами, например, переименовать их или удалить из проекта. Здесь могут появляться и другие типы файлов, например, ресурсы проектов (ресурсами называются файлы, используемые проектом, но не являющиеся файлами C#, допустим, двоичные файлы изображений или звуковые файлы). Еще раз повторим, что мы можем работать с ними с помощью того же самого интерфейса.

Вхождение References (ссылки) содержит список тех библиотек, которые используются в данном проекте. Это тоже относится к тому, что мы собираемся рассматривать ниже, поскольку пока нас вполне удовлетворяют стандартные ссылки.

Во втором окне — Class View — находится альтернативное представление проекта, которое позволяет увидеть структуру созданного кода. Мы вернемся к рассмотрению этого окна позднее, а сейчас вам достаточно знать, что Solution Explorer позволяет выбирать представление проекта по нашему усмотрению.



Если вы попробуете щелкать мышью по другим файлам и иконкам в этих окнах, то обнаружите, что содержимое нижнего окна при этом меняется. Оно также имеет различные внешние представления, но самым важным для нас является представление Properties (см. рис. справа).

В этом окне выводится дополнительная информация о том, что мы выбираем в окне Solution Explorer/Class View, расположенном над ним. Так, например, вид, изображенный на приведенном выше рисунке, получается при выборе файла `Class1.cs`. В этом окне может появляться информация и о других объектах, которые могут быть выбраны,— например, о компонентах пользовательского интерфейса, с чем мы встретимся в разделе, посвященном созданию оконного приложения.

Часто изменения, вносимые в различные вхождения в окне Properties, оказывают самое непосредственное влияние на наш код, приводя к появлению новых строк кода или к изменению уже существующих. При написании некоторых проектов мы будем затрачивать на манипуляции с этим окном примерно столько же времени, сколько и на изменения, вносимые в код вручную.

Теперь давайте посмотрим на окно Output. Когда мы выполняли нашу программу, вы, возможно, обратили внимание на текст, появившийся в этом окне перед тем, как нашим приложением было создано консольное окно. На компьютере автора было выведено следующее:

```
----- Build started: Project: ConsoleApplication1, Configuration: Debug .NET -----
        // Построение начато: Проект: ConsoleApplication1, конфигурация: Debug .NET //

Preparing resources...                          // Подготовка ресурсов... //
Updating references...                          // Обновление ссылок... //
Performing main compilation...                  // Выполнение основной компиляции... //

Build complete - 0 errors, 0 warnings
                                  // Построение завершено -- ошибок 0, предупреждений 0 //
Building satellite assemblies...      // Построение сопутствующих модулей... //


------------------- Done ----------------------                    // Выполнено //

    Build: 1 succeeded, 0 failed, 0 skipped
                            // Построено: успешно 1, неудачно 0, пропущено 0 //
    Deploy: 0 succeeded, 0 failed, 0 skipped
                            // Установлено: успешно 0, неудачно 0, пропущено 0 //
```

Как вы, вероятно, уже догадались, это отчет о компиляции наших файлов. В этом же отчёте будут размещаться сообщения обо всех ошибках, которые возникли в процессе компиляции. Для наглядности попробуйте убрать точку с запятой из строки, которую мы добавили в предыдущем разделе, и откомпилировать программу заново. На этот раз вы увидите следующее:

```
----- Build started: Project: ConsoleApplication1, Configuration: Debug .NET -----

Preparing resources...
Updating references...
Performing main compilation...
c:\begcsharp\chapter2\consoleapplication1\class1.cs(15,73): error CS1002: ; expected
                                                    // ошибка CS1002: требуется ; //

Build complete -- 1 errors, 0 warnings
Building satellite assemblies...

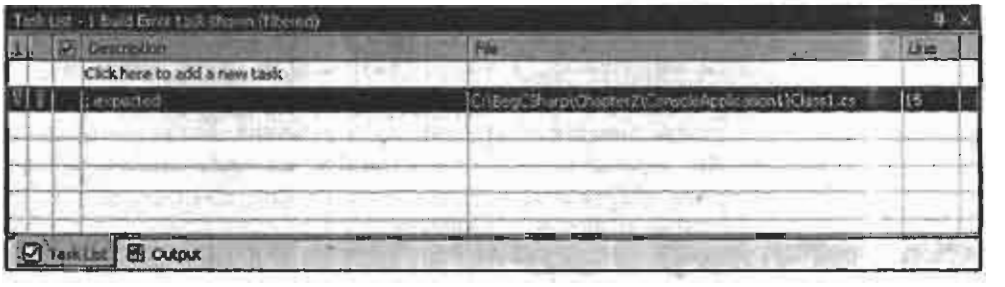

---------------------- Done ----------------------

    Build: 0 succeeded, 1 failed, 0 skipped
    Deploy: 0 succeeded, 0 failed, 0 skipped
```

В этом случае нам не удалось запустить приложение.

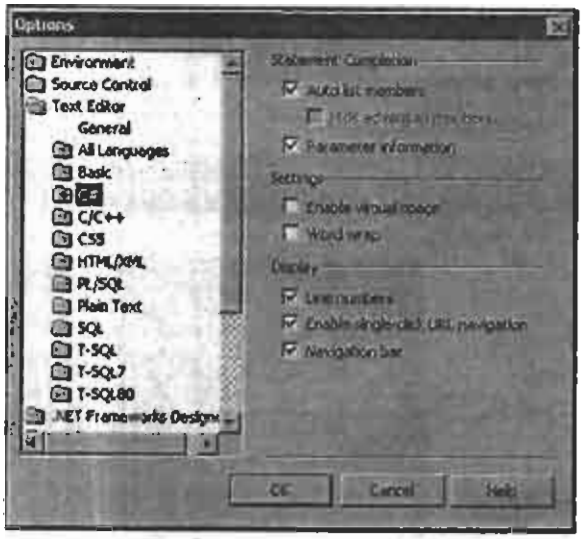
> *В следующей главе, когда мы приступим к рассмотрению синтаксиса C#, вы узнаете, что наличие точки с запятой требуется почти всюду — в конце практически каждой строки кода.*

Теперь, когда нужно предпринять некоторые действия для того, чтобы заставить программу работать, VS автоматически добавил это задание в список заданий, который располагается в том же окне Output:



Это окно помогает исправлять ошибки в коде, поскольку в нем выводятся сообщения о том, что необходимо сделать для успешной компиляции проекта. Если два раза щелкнуть мышью на ошибке, указанной в данном окне, то курсор переместится на ту позицию в строке исходного кода, где ошибка содержится (если файл с исходным текстом не был открыт ранее, он откроется в этот момент); это позволяет нам быстро исправлять наши промахи. В местах, где обнаружены ошибки, также можно увидеть волнистые линии красного цвета, благодаря чему можно быстро просматривать исходный код для выявления ошибок.

Обратите внимание на то, что местонахождение ошибки указывается в виде номера строки. В текстовом редакторе VS номера строк по умолчанию не выводятся, однако такой режим вполне достоин того, чтобы в него перейти. Для этого потребуется установить соответствующий флажок в диалоговом окне Options (опции), в которое можно перейти через пункт меню Tools | Options.... Это контрольное



окошко называется Line Numbers (номера строк) и находится в разделе Text Editor | C# | General, как показано на рисунке слева.

С помощью этого диалогового окна можно получить доступ ко многим полезным опциям; некоторыми из них мы будем пользоваться в дальнейшем.

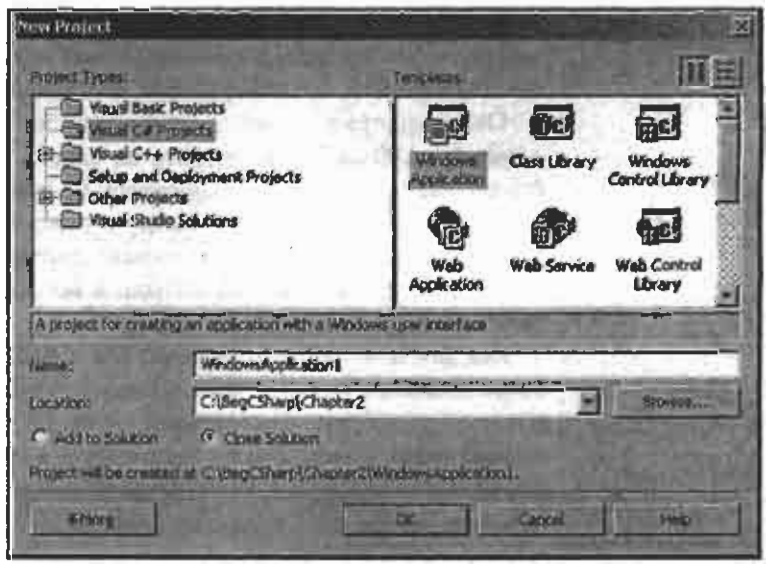
Приложения Windows Forms

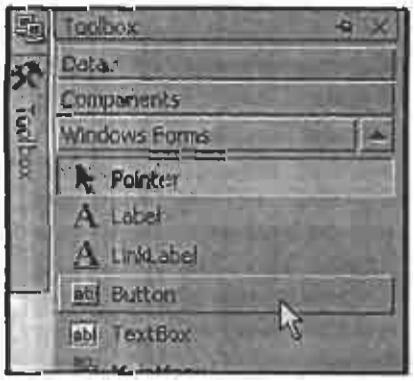
Часто оказывается удобнее продемонстрировать работу кода, запуская его как часть оконного приложения, а не через окно консоли или через командную строку. Достичь этого можно, воспользовавшись отдельными "строительными блоками" для создания пользовательского интерфейса.

Сейчас мы познакомимся с основами этого процесса. Вы узнаете, как создать и запустить оконное приложение, но в подробности относительно того, что это приложение выполняет на самом деле, мы вдаваться не будем. Ниже мы вернемся к детальному рассмотрению оконных приложений.

Практикум: создание простого оконного приложения

1. Создайте новый проект типа Windows Application в той же директории, что и предыдущий (`C:\BegCSharp\Chapter2`), с присвоенным ему по умолчанию именем `WindowsApplication1`. Если первый проект по-прежнему открыт, то для того, чтобы начать новый проект, воспользуйтесь опцией Close Solution (закрыть решение) (см. рис. справа).





2. Переместите указатель мыши сначала на панель Toolbox, расположенную в левой части экрана, затем на вхождение Button (кнопка) вкладки Windows Forms, и дважды щелкните левой кнопкой мыши: кнопка добавится в основную форму приложения (`Form1`) (см. рис. слева).

3. Два раза щелкните мышью по кнопке, добавленной в форму.

4. На экране должен появиться код на C#, который находится в этой форме. Внесите в него следующие изменения (для краткости здесь приводится только часть того кода, который хранится в файле):

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

5. Запустите приложение.

6. Нажмите появившуюся кнопку, и откроется диалоговое окно с сообщением (см. рис. справа).



Как это работает

Повторим еще раз очевидное: VS выполняет за нас огромный объем работы, существенно упрощая создание функциональных оконных приложений и экономя наши усилия. Созданное приложение ведет себя так же, как и другие окна: его можно перемещать, закрывать, минимизировать и т. п. Нам не потребовалось писать для этого никакого кода — все и так работает. То же самое справедливо и для кнопки, которую мы поместили в окно. Достаточно было два раза щелкнуть по ней мышью, чтобы VS понял: мы желаем иметь код, который должен выполняться при нажатии на эту кнопку. Все, что от нас потребовалось,— это предоставить необходимый код, в итоге совершенно бесплатно получив полноценную функциональную работающую кнопку.

Оконные приложения, естественно, не ограничиваются простыми формами с кнопками. Если вы посмотрите на панель с инструментами, из которой мы выбрали опцию Button, то обнаружите большое количество самых разнообразных блоков, с помощью которых можно строить пользовательский интерфейс; некоторые из них могут быть вам знакомы, другие — нет. В этой книге будет описано применение почти всех этих блоков, и вы увидите, насколько просто их использовать и сколько времени и сил это позволяет сэкономить.

Код для приложения, находящийся в файле `Form1.cs`, производит впечатление более сложного, чем код, использовавшийся в предыдущем разделе, однако так кажется лишь потому, что во втором случае он немного длиннее. Большая часть кода, находящегося в этом файле, относится к внешнему представлению управляющих элементов в форме; именно благодаря этому коду приложение можно просматривать в основном окне в режиме разработки (этот процесс является визуализацией данного кода). Кнопка — это один из примеров управляющих элементов, ее можно использовать наряду с другими строительными блоками пользовательского интерфейса, находящимися в разделе Windows Forms панели Toolbox.

Давайте более тщательно рассмотрим кнопку в качестве образца управляющего элемента. Вернитесь обратно в Design View (режим разработки) формы, воспользовавшись соответствующей вкладкой основного окна, и выберите кнопку, один раз щелкнув по ней мышью. После этого в окне Properties, расположенном в нижней правой части экрана, появятся свойства кнопки, используемой нами в качестве управляющего элемента (управляющие элементы обладают свойствами, во многом напоминающими свойства файлов, которые рассматривались в предыдущем примере).



С помощью полосы прокрутки перейдите к свойству `Text`, которое в этот момент имеет значение button1, и замените его на Click Me (см. рис. справа).

Это приведет к тому, что текст, находящийся на кнопке в форме `Form1`, соответствующим образом изменится.

У данной кнопки имеется много свойств, начиная от простых, определяющих ее цвет и размер, и заканчивая менее понятными установками, например, привязками к данным, которые позволяют вносить ссылки на базы данных. Как уже упоминалось при разборе предыдущего примера, изменение свойств зачастую приводит к непосредственным изменениям в коде, и данный случай не является исключением. Вернитесь в режим просмотра кода `Form1.cs`: сейчас мы проанализируем те изменения, которые только что в него внесли.



При поверхностном осмотре вы, возможно, вообще не обнаружите в программе никаких изменений. Это следствие того, что разделы кода, отвечающие за внешний вид и форму управляющих элементов, скрыты от наших глаз (в конце концов вряд ли имеет смысл рассматривать код, если можно увидеть результат его выполнения).

Для решения этой задачи VS использует **систему выделения кода**. Это можно увидеть на рисунке слева.

Если вы посмотрите влево от кода, то возле номеров строк (если задана опция их вывода) вы обнаружите некие серые линии и квадратики, в которых содержатся символы + и –. Эти квадратики используются для разворачивания и сворачивания определенных участков программы. Приблизительно в середине файла (в файле автора это строка 48, но ее номер может меняться) располагается квадратик, в котором находится символ +, а в теле самой программы имеется помещенная в рамку строка **"Windows Form Designer generated code"** ("Код, сгенерированный Windows Form Designer"). Смысл этих слов следующий: "Здесь находится некоторый код, сгенерированный VS, знать о котором вам совершенно не обязательно". Однако при желании мы можем его просмотреть и увидеть, что именно произошло при изменении свойств управляющего элемента. Просто щелкните мышью по квадрату, в котором находится символ +, и код станет видимым. Где-нибудь внутри этого кода вы обнаружите следующую строку:

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

Не вчитываясь в сам код, мы смогли убедиться, что текст, набранный в окне **Properties**, попал непосредственно в программу.

Такой способ выделения чрезвычайно удобен при написании кода, поскольку он позволяет разворачивать и сворачивать самые разные участки программы, а не только те, которые в обычном режиме скрыты от нас. Точно так же, как оглавление книги позволяет получить общее представление о ее содержании, просмотр последовательности свернутых участков кода способствует быстрому поиску в больших программах на C#.

Итоги

В данной главе вы познакомились с некоторыми инструментами, которые будут использоваться в остальной части книги. Мы — в общих чертах — рассмотрели среду разработки Visual Studio.NET и использовали ее для создания двух типов приложений. Более простое из них — консольное приложение; его вполне достаточно для решения большинства наших задач, и оно позволяет нам сосредоточиться на основах программирования на C#. Оконные приложения несколько сложнее, однако визуально они оказываются более впечатляющими и наглядными для пользователей, знакомых с оконной средой (т. е. почти для всех).

Теперь, когда вы научились создавать простые приложения, можно перейти к настоящему изучению C#. Следующая глава посвящена основам синтаксиса C# и структуре программ на C#, которые необходимо освоить, прежде чем переходить к более сложным объектно-ориентированным методам. А изучив их, мы сможем перейти к вопросу об использовании C# для получения доступа к тем большим возможностям, которые предоставляет .NET Framework.



Переменные и выражения

Возможно, наиболее фундаментальное определение компьютерной программы — "последовательность операций над данными". Это утверждение оказывается справедливым даже для наиболее сложных случаев, таких, например, как развернутые, многофункциональные Windows-приложения типа Microsoft Office Suite. И хотя пользователи зачастую не задумываются над структурой приложений, внутренняя сущность последних заключается именно в этом.

Попробуем проиллюстрировать сказанное на примере обычного компьютерного монитора. Когда вы смотрите на экран, трудно представить что-то кроме "движущейся картинки". Хотя на самом деле то, что вы видите, является всего лишь представлением некоторых данных, которые — в своем исходном виде — просто последовательность нулей и единиц, хранящихся где-то внутри компьютера. Какие бы действия вы ни выполняли на экране — перемещали бы курсор, щелкали мышью по иконке или вводили текст в текстовом процессоре, — все это приводит к определенным изменениям данных, хранящихся в памяти.

Можно привести и не столь абстрактный пример. Когда вы используете приложение-калькулятор, вы вводите данные, представленные в форме чисел, и выполняете над этими числами различные операции почти так же, как с помощью карандаша и бумаги, только намного быстрее!

Итак, если фундаментальное свойство компьютерных программ заключается в выполнении операций над данными, то это предполагает наличие, во-первых, некоторого способа хранения данных и, во-вторых, определенных методов для выполнения манипуляций над ними. Для обеспечения этих двух функций используются соответственно **переменные** и **выражения**, и данная глава посвящена их изучению.

Вначале, однако, нам придется рассмотреть базовый синтаксис, используемый при программировании на C#.

Базовый синтаксис C#

Внешне код на C# очень напоминает код на C++ или на Java. На первый взгляд этот синтаксис может показаться весьма запутанным и гораздо меньше похожим на обыкновенный письменный английский, чем некоторые другие языки. Однако, погрузившись в мир программирования на C#, вы обнаружите, что используемый в нем стиль является весьма разумным, а написание удобочитаемых программ не представляет никаких сложностей.

В отличие от компиляторов с некоторых других языков программирования, компиляторы C# не обращают внимание на лишнее пустое пространство, которое может состоять из пробелов, возвратов каретки или символов табуляции (эти символы

известны под общим названием **символов пустого пространства**). Отсюда можно сделать вывод, что мы обладаем большой свободой в выборе способа форматирования нашей программы, хотя следование определенным правилам помогает создавать более удобные для чтения программы.

Код на C# представляет собой **последовательность** операторов, каждый из которых оканчивается точкой с запятой. Поскольку пустое пространство игнорируется, то можно располагать по несколько операторов в одной строке; однако для того чтобы сделать программу более наглядной, принято размещать символ возврата каретки после каждой точки с запятой, в результате чего на строке располагается только один оператор. Хотя абсолютно приемлемо (и совершенно нормально) использовать операторы, которые занимают несколько строк кода.

C# — язык, обладающий **блочной структурой**; другими словами, каждый оператор является частью некоторого **блока** кода. Эти блоки, для обозначения начала и конца которых используются фигурные скобки { и }, могут содержать произвольное количество операторов или не содержать их вовсе. Обратите внимание, что фигурные скобки не сопровождаются точками с запятой.

Итак, простой блок программы на C# может иметь следующий вид:

```
{
   <строка кода 1, оператор 1>;
   <строка кода 2, оператор 2>
      <строка кода 3, оператор 2>;
}
```

Здесь выражения *<строка кода x, оператор y>* не являются действительными составляющими кода на C#: этот текст используется для обозначения места, куда впоследствии будут помещены операторы C#. Обратите внимание, что в данном примере вторая и третья строки кода являются частями одного и того же оператора, поэтому после второй строки точка с запятой отсутствует.

В этом простом примере кода использованы **отступы**, чтобы сделать запись на C# более наглядной. Такой подход не является изобретением автора — это обычная практика; в действительности VS автоматически делает такие отступы по умолчанию. Каждый блок кода имеет свою **величину** отступов, определяющую, насколько далеко вправо он сдвинут. Блоки могут быть **вложены** друг в друга (т. е. блоки могут содержать в себе другие блоки), и в таком случае величина отступа у вложенных блоков должна быть больше:

```
{
   <строка кода 1>;
   {
      <строка кода 2>;
      <строка кода 3>;
   }
   <строка кода 4>;
}
```

Кроме того, больший отступ обычно используется для тех строк кода, которые являются продолжением предшествующей строки, как в случае строки 3 в первом из приведенных примеров.

Запомните: такой стиль ни в коем случае не является обязательным. Однако если вы не будете его использовать, то по мере прочтения этой книги вы будете путаться все сильнее и сильнее!

Еще одна вещь, которая может часто встречаться в коде на C#, это **комментарии**. Строго говоря, комментарии не являются частью кода на C#, однако они спокойно сосуществуют с ним. Предназначение комментариев в точности соответствует

их названию: они позволяют включать в программу описательный текст на обычном английском (французском, немецком, монгольском и т. д.) языке, который игнорируется компилятором. Когда мы приступаем к созданию достаточно длинных сегментов кода, то бывает полезно включить в программу напоминание о том, что именно мы собираемся здесь сделать, например: "в этой строке кода у пользователя запрашивается число" или "этот сегмент кода написан Бобом". В C# предусмотрены два способа включения комментариев. Можно использовать либо специальные маркеры, обозначающие начало и завершение комментария, либо маркер, который означает, что "все, что находится за ним до конца строки, является комментарием". Последний способ является исключением из правила, о котором мы упоминали ранее и которое гласит, что компиляторы C# игнорируют все символы возврата каретки, но это особый случай.

Для того чтобы обозначить комментарии по первому способу, мы будем использовать сочетания символов `/*` в начале комментария и `*/` в конце комментария. Они могут располагаться как на одной строке, так и на разных строках; в последнем случае все строки, находящиеся между ними, также являются частью комментария. Единственное, что не может входить в состав комментария,— это сочетание символов `*/`, поскольку оно будет интерпретировано как конец комментария. Итак, вполне допустимы следующие комментарии:

```
/* Это комментарий */

/* Так же, как ...

                        ... и это! */
```

А вот приведенный ниже пример вызовет проблемы:

```
/* Сочетание символов **/* часто используется для завершения комментария */
```

В этом случае последняя часть комментария (символы, расположенные после `*/`) будет интерпретирована как код C#, что приведет к возникновению ошибки.

Другой подход к оформлению комментария состоит в использовании в качестве его начала символов `//`. В следующем за ними комментарии мы можем написать все, что угодно... но не более одной строки! Данный пример является правильным:

```
// Это еще один вид комментария.
```

А вот попытка воспользоваться следующим комментарием приведет к неудаче, поскольку вторая строка будет интерпретироваться как код на C#:

```
// Это тоже комментарий,
   а вот это уже нет.
```

Такой вид комментариев оказывается полезным для документирования отдельных операторов, поскольку весь комментарий целиком умещается на одной строке:

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

В C# существует еще и третий тип комментариев, который позволяет создавать документацию для программы. Этот комментарий также расположен на одной строке, но в отличие от последнего случая начинается с трех символов `/`, а не с двух:

```
/// Это особый вид комментария
```

В обычных условиях эти комментарии игнорируются компилятором, как и все остальные, однако можно настроить VS таким образом, что текст этих комментариев при компиляции проекта будет извлекаться и использоваться для создания

специальным образом отформатированного текстового файла. На его основе в дальнейшем можно создавать документацию по проекту. Этот вопрос будет детально рассматриваться в главе 18.

Чрезвычайно важным моментом, который необходимо учитывать при работе с C#, является **чувствительность** этого языка **к регистру**. В отличие от некоторых других языков программирования, на C# текст необходимо вводить, в точности соблюдая регистр, поскольку простая замена заглавной буквы на строчную приведет к прекращению компиляции проекта.

Базовая структура консольного приложения на C#

Давайте еще раз обратимся к примеру консольного приложения из предшествующей главы (`ConsoleApplication1`) и потратим некоторое время на разбор структуры программы. Код приложения имеет следующий вид:

```
using System;

namespace ConsoleApplication1
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      static void Main(string[] args)
      {
         //
         // TODO: Add code to start application here
         //       static void Main(string[] args)
         Console.WriteLine("The first app in Beginning C# Programming!");
      }
   }
}
```

Нетрудно заметить, что в этом приложении присутствуют все синтаксические элементы, обсуждавшиеся выше. Мы можем обнаружить здесь точки с запятыми, фигурные скобки, комментарии и отступы.

Наиболее важной частью кода, насколько мы можем судить об этом на данный момент, является следующая:

```
static void Main(string[] args)
{
   //
   // TODO:  Add code to start application here
   //       static void Main(string[] args)

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

Это тот код, который выполняется при запуске консольного приложения, или, если говорить немного точнее, тот заключенный в фигурные скобки блок кода, который и выполняется при запуске. Единственной строкой, выполняющей какие-либо действия, является строка, которую мы добавили к автоматически сгенерированному коду и которая — одна из всех — не является комментарием. Этот код выводит некоторый текст в консольном окне, а точный механизм его работы нас пока не интересует.

Поскольку основной задачей первых двух глав является объяснение базового синтаксиса C#, понимание того, каким образом выполнение приложения доходит до точки вызова `Console.WriteLine()`, сейчас не важно. Значимость этого дополнительного кода станет понятна позднее.

Переменные

Как упоминалось во введении к этой главе, переменные тесно связаны с хранением данных. По существу, переменные в памяти компьютера можно представить в виде коробочек, лежащих на полке. Такие коробочки могут использоваться для того, чтобы что-нибудь в них класть, а затем вынимать обратно, или же можно просто заглянуть в них и узнать, есть там что-то или нет. Все это относится и к переменным, в которые мы можем помещать данные, а затем брать их оттуда или просматривать по мере необходимости.

Хотя все данные в компьютере фактически представляют собой одно и то же (последовательность, состоящую из нулей и единиц), переменные могут отличаться друг от друга, образуя так называемые типы. Возвращаясь к нашей аналогии с коробочками, можно представить коробочки различных размеров и форм, причем разные объекты могут помещаться только в специально предназначенные для них коробочки. Обоснование системы типов заключается в том, что для манипуляций над данными различных типов могут требоваться различные методы, и, приписав переменным конкретные типы, мы можем избежать серьезной путаницы. Ведь вряд ли получится что-то осмысленное, если рассматривать последовательность нулей и единиц, представляющую собой оцифрованное изображение, в качестве звукового файла.

Для того чтобы использовать переменную, необходимо ее **объявить**. Это означает, что ей должны быть присвоены имя и тип. Как только переменная объявлена, она может использоваться как хранилище для данных того типа, который указан в ее объявлении.

Синтаксис объявления переменных в C# состоит из указания типа переменной и ее имени:

```
<тип> <имя>;
```

Если попытаться использовать переменную, которая не была объявлена, то программа не будет откомпилирована. Однако в этом случае компилятор сможет точно указать, в чем именно заключается проблема, и поэтому такой тип ошибок не очень опасен. Попытка использовать переменную, которой не присвоено никакого значения, тоже приведет к ошибке; в этом случае компилятор также сможет точно установить причину ее возникновения.

Итак, какие типы можно использовать?

На самом деле существует бесконечное множество типов, которые можно использовать, так как мы имеем возможность описывать свои собственные типы, предназначенные для хранения любых самых "хитрых" данных.

Однако существуют определенные типы данных, которые раньше или позже потребуются всем, например, переменные, в которых могут храниться числа. По этой причине существует некоторое количество простых, предварительно описанных типов данных, с которыми мы сейчас и начнем знакомиться.

Простые типы

Простые типы — это такие типы значений, как численные или логические (true (истина) или false (ложь)), которые являются фундаментальными составными частями любых приложений, а также других, более сложных типов. Большинство из имеющихся в нашем распоряжении типов являются численными, что на первый взгляд выглядит несколько странно: разве для хранения чисел не достаточно одного типа?

Причина такого изобилия числовых типов кроется в способе хранения чисел в памяти компьютера как последовательности нулей и единиц. В случае целых значений мы используем несколько бит (отдельных цифр, которые могут принимать значение нуля или единицы) и просто представляем число в двоичном формате. Переменная, в которой может храниться N бит, позволяет представлять любые числа в диапазоне от 0 до $(2^N - 1)$. Числа, превышающие последнее значение, оказываются слишком большими и не могут быть размещены в переменной данного типа.

В качестве примера давайте рассмотрим переменную, в которой может храниться 2 бита. Соответствие между целыми числами и их двоичным представлением в этом случае будет следующим:

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

Если нам необходимо хранить бóльшие числа, то для этого потребуется большее количество бит (3 бита, например, позволят хранить числа, лежащие в диапазоне от 0 до 7).

Отсюда неизбежно следует вывод о том, что хранение произвольного числа требует бесконечного количества бит, что, безусловно, неприемлемо для персонального компьютера. Но даже если бы необходимое количество бит и имелось, использовать такой объем памяти для числа, изменяющегося, скажем, от 0 до 10, было бы крайне неэффективно (из-за неиспользуемой памяти). В данном случае вполне достаточно 4 бит, которые позволяют разместить в одной и той же области памяти все числа указанного диапазона.

Именно для этого существует несколько целых типов, которые могут быть использованы для хранения чисел из различных диапазонов и которые занимают различный объем памяти (вплоть до 64 битов). Эти типы целых чисел сведены в таблицу, приводимую ниже.

Обратите внимание, что каждый тип использует один из стандартных типов, определенных в .NET Framework. Как уже упоминалось в главе 1, применение стандартных типов — это как раз то, что делает возможным совместное использование различных языков программирования. Имена, которые используются для этих типов в C#, обозначают определенные типы, описанные в .NET Framework. В таблице приводятся имена типов и указывается, какому типу в библиотеке .NET Framework они соответствуют.

Тип	Тип .NET Framework	Допустимые значения
sbyte	System.SByte	Целое в диапазоне от –128 до 127
byte	System.Byte	Целое в диапазоне от 0 до 255
short	System.Int16	Целое в диапазоне от –32 768 до 32 767
ushort	System.UInt16	Целое в диапазоне от 0 до 65 535
int	System.Int32	Целое в диапазоне от –2 147 483 648 до 2 147 483 647
uint	System.UInt32	Целое в диапазоне от 0 до 4 294 967 295
long	System.Int64	Целое в диапазоне от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
ulong	System.UInt64	Целое в диапазоне от 0 до 18 446 744 073 709 551 615

Буква "u", встречающаяся перед некоторыми из имен, является сокращением от слова "unsigned" ("без знака"). Это говорит о том, что в переменных соответствующего типа не могут храниться отрицательные значения. Этот факт отражен в столбце "Допустимые значения".

Естественно, что кроме целых чисел нам требуется хранить значения с **плавающей запятой**, которые представляют нецелые числа. Существует три типа переменных с плавающей запятой, которые мы можем использовать: `float`, `double` и `decimal`. Первые два из них хранят числа с плавающей запятой в виде $+/-m \times 2^e$, причем диапазон допустимых значений `m` и `e` для каждого типа свой. Тип `decimal` использует альтернативную форму представления чисел $+/-m \times 10^e$. Эти три типа приводятся ниже вместе с допустимыми значениями `m` и `e` и диапазонами действительных чисел:

Тип	Тип .NET Framework	Минимальное значение m	Максимальное значение m	Минимальное значение e	Максимальное значение e	Приблизительное минимальное число	Приблизительное максимальное число
`float`	`System.Single`	0	224	–149	104	$1,5 \times 10^{-45}$	$3,4 \times 10^{38}$
`double`	`System.Double`	0	253	–1075	970	$5,0 \times 10^{-324}$	$1,7 \times 10^{308}$
`decimal`	`System.Decimal`	0	296	–26	0	$1,0 \times 10^{-28}$	$7,9 \times 10^{28}$

Кроме численных типов, существует еще три простых типа:

Тип	Тип .NET Framework	Диапазон допустимых значений
`char`	`System.Char`	Отдельный символ в кодировке Unicode, хранящийся в виде целого числа в диапазоне от 0 до 65 535
`bool`	`System.Boolean`	Логическое значение (Boolean), которое может принимать значения `true` (истина) или `false` (ложь)
`string`	`System.String`	Последовательность символов

Обратите внимание, что верхней границы количества символов, составляющих строки типа `string`, не существует, поскольку они могут занимать изменяющиеся объемы памяти.

Логический тип `bool` — один из наиболее часто используемых в C# типов переменной; аналогичные типы часто встречаются в кодах на других языках программирования. Использование переменной, которая может принимать только два значения — либо `true`, либо `false`,— создает очень важную возможность ветвления логики приложения. В качестве простого примера представьте себе, какое огромное количество вопросов допускают в качестве ответа `true` или `false` ("да" или "нет"). Сравнение значений переменных и проверка допустимости введенной информации как раз представляют собой примеры программного использования логических переменных (эти примеры вскоре станут предметом нашего изучения).

Теперь, когда мы познакомились с этими типами, давайте рассмотрим небольшой пример их объявления и использования.

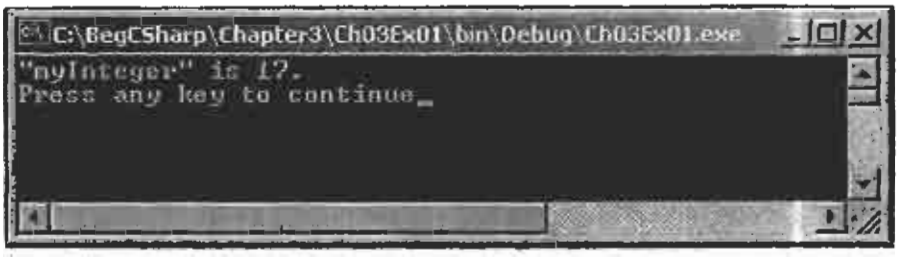
Практикум: использование переменных простых типов

1. Создайте новое консольное приложенне с именем `Ch03Ex01` в директории `C:\BegCSharp\Chapter3`.

2. Добавьте следующий код в `Class1.cs` (и уберите строки с комментариями):

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

3. Выполните программу (не забудьте, что прн выполнении программы в режиме отладки окно консольного приложения закроется быстрее, чем вы сможете понять, что произошло):



Как это работает

Код, который мы ввели, выполняет три действия:

- Объявляет две переменные.
- Присваивает этим переменным значения.
- Выводит значения этих двух перемеиных на консоль.

Объявление переменных осуществляет следующий код:

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

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

> *Запомните, что выбор имен для переменных имеет свои ограничения и нельзя использовать любую произвольную последовательность символов. Этот вопрос будет рассмотрен в разделе, посвященном именованию переменных.*

В следующих двух строках кода происходит присваивание значений:

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

В данном случае происходит присваивание перемеиным двух фиксированиых значений (известных под названием **литеральных значений**) посредством оператора **присваивання =** (более близко мы познакомимся с операторами в разделе "Выражеиня" этой главы). Мы присваиваем целое значение 17 переменной `myInteger`,

а строку `"myInteger" is` (включая кавычки) переменной `myString`. Обратите внимание, что когда мы присваиваем строковые литеральные значения таким способом, строка должна обрамляться двойными кавычками. При этом существуют определенные символы (например, символ двойных кавычек), включение которых в строку может привести к возникновению некоторых проблем; поэтому необходимо избегать появления ряда символов путем их замены на некоторую последовательность символов (так называемую escape-последовательность), которая обозначает символ, который нам нужен. В данном примере мы используем последовательность символов `\"` для того, чтобы избежать появления двойных кавычек внутри переменной:

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

Если бы мы не стали пользоваться этими escape-последовательностями и попытались бы записать данный оператор следующим образом:

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

это привело бы к ошибке при компиляции.

Здесь необходимо обратить внимание на то, что присваивание литеральных строк — это еще одна ситуация, при которой необходимо соблюдать аккуратность в использовании символов конца строки: компилятор C# не воспринимает литеральные строки, располагающиеся более чем на одной строке программы. Если нам требуется включить в строку символ конца строки, то в этом случае мы можем воспользоваться escape-последовательностью для символа возврат каретки `\n`. Например, полученная в результате следующего присваивания строка:

```
myString = "Данная строка содержит \nсимвол конца строки.";
```

будет выводиться на консоли в виде двух строк:

```
Данная строка содержит
символ конца строки.
```

Любая escape-последовательность состоит из символа обратного слэша, за которым следует еще один символ из весьма ограниченного множества символов (с этим множеством мы познакомимся немного позднее). Поскольку символ обратного слэша используется для такой специфической цели, то для него существует своя собственная escape-последовательность, которая представляет собой просто последовательность из двух обратных слэшей — `\\`.

Возвращаясь к нашему коду, мы видим еще одну строку, которую пока еще не рассматривали:

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

Эта строка напоминает тот простой метод вывода текста на консоль, которым мы воспользовались в нашем первом примере, но на этот раз мы выводим переменные. Поскольку нет необходимости бежать впереди паровоза, на данном этапе мы не будем слишком глубоко вдаваться в подробности, касающиеся этой строки кода. Достаточно сказать, что этим способом мы будем пользоваться на протяжении всей первой части книги для вывода текста в консольное окно. Внутри скобок имеются два элемента:

- Строка
- Список переменных, значения которых мы хотим включить в выходную строку, разделенные запятыми

Строка, которую мы выводим — "{0} {1}.", — не производит впечатления очень полезного текста. Однако, как вы можете заметить, это вовсе не тот текст, который выводится при запуске приложения. Причина заключается в том, что эта строка в действительности является неким шаблоном, в который вставляется содержимое переменной. Каждый комплект фигурных скобок в этой строке представляет собой зарезервированное место, в котором будет размещаться содержимое всех переменных, перечисленных в списке. Каждое такое зарезервированное место (или форматная строка) имеет вид целого числа, заключенного в фигурные скобки. Эти целые числа имеют значения, начинающиеся с 0 и увеличивающиеся на 1, а суммарное количество таких мест должно соответствовать общему числу переменных, указанных в следующем за строкой списке через запятую. Когда происходит вывод текста на консоль, в каждое из зарезервированных мест помещается значение соответствующей переменной. В примере, приведенном выше, {0} заменяется на фактическое значение первой переменной — `myString`, а {1} заменяется на содержимое `myInteger`.

Этим методом мы впредь будем пользоваться для вывода на консоль текста из наших следующих программ.

Именование переменных

Как уже было сказано в предшествующем разделе, мы не можем использовать в качестве имени переменной произвольную последовательность символов. Однако это не так страшно, как может показаться на первый взгляд, поскольку в нашем распоряжении имеется весьма гибкая система именования.

Основные правила при именовании переменных следующие:

- Первый символ в имени переменной должен быть либо буквой, либо символом подчеркивания _, либо @.
- Последующими символами могут быть буквы, символ подчеркивания или цифры.

Кроме того, существуют определенные ключевые слова, которые обладают особым смыслом с точки зрения компилятора C#, например, `using` или `namespace`. Если вы по ошибке используете одно из них, компилятор выразит свое недовольство, о чем вы непременно вскоре узнаете, поэтому об этом также не следует слишком волноваться.

Следующие имена переменных являются правильными:

```
myBigVar
VAR1
_test
```

А вот эти — нет:

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

И не забывайте о том, что C# чувствителен к регистру: необходимо постоянно помнить о правильном использовании регистров в соответствии с тем, как они использовались при объявлении переменных. Обращение к переменной внутри программы, в котором хотя бы один символ будет набран в неверном регистре, приведет к прекращению компиляции.

Другим следствием этой ситуации является возможность иметь множество имен переменных, которые будут отличаться друг от друга только использованием регистра; например, приведенные ниже имена различны:

```
myVariable
MyVariable
MYVARIABLE
```

Соглашения, используемые при именовании

Имена переменных — это нечто такое, что приходится использовать *очень часто*. Поэтому стоит посвятить немного времени тому, какие типы имен следует применять. Однако прежде чем перейти к обсуждению этого вопроса, следует отметить, что данная тема полна противоречий. На протяжении многих лет одни системы именования сменялись другими, а некоторые разработчики готовы насмерть стоять за признание только собственных систем.

До недавнего времени наиболее популярной была система, известная под именем **венгерской системы записи**. Она предполагает использование перед именами всех переменных префикса, набираемого в нижнем регистре и обозначающего тип переменной. Например, если переменная имеет тип `int`, то перед ней следует поместить символ `i` (или `n`), например, `iAge`. При использовании такой системы оказывается очень просто определять тип переменных даже при беглом взгляде на программу.

Однако в более современных языках, вроде C#, реализовать венгерскую систему оказывается совсем не просто. Эта система вполне может использоваться для тех типов, с которыми вы успели познакомиться, поскольку для обозначения каждого типа вполне достаточно префиксов, состоящих из одной или двух букв. Но поскольку у нас имеется возможность создавать свои собственные типы и, кроме того, в базовой .NET Framework таких сложных типов присутствует несколько сотен, то система быстро перестает работать. Когда над проектом трудятся несколько человек, существенно возрастает вероятность того, что с использованием различных префиксов будет возникать путаница, которая может иметь самые разрушительные последствия.

В итоге разработчики пришли к выводу, что намного лучше именовать переменные просто в соответствии с их предназначением. Если при этом возникают какие-либо сомнения, то узнать тип переменной очень просто. В VS для этого надо просто поместить курсор мыши около имени переменной, и очень быстро появится окно, в котором будет указан ее тип.

В настоящее время в пространствах имен .NET Framework используются два соглашения об именовании, известные как **PascalCase** и **camelCase**. Отличить одну систему от другой можно по использованию регистра. Обе системы применяются к именам, составленным из нескольких слов; при этом каждое слово в составном имени набирается в нижнем регистре, за исключением первой буквы слова, которая должна набираться в верхнем регистре. В системе camelCase существует еще одно дополнительное правило, которое гласит, что первое слово должно начинаться с буквы в нижнем регистре.

Следующие имена переменных соответствуют правилам camelCase:

```
age
firstName
timeOfDeath
```

А вот так имена должны записываться в соответствии с правилами PascalCase:

```
Age
LastName
WinterOfDiscontent
```

В случае простых переменных мы будем следовать системе camelCase, а при составлении некоторых более сложных имен мы будем использовать систему PascalCase, что соответствует рекомендациям компании Microsoft.

Наконец, необходимо отметить, что многие предыдущие системы именования широко использовали символ подчеркивания, обычно в качестве разделителя слов в имени переменной: например, `my_first_variable`. Такой подход в настоящее время не рекомендуется (чему автор очень рад, поскольку всегда считал, что это выглядит уродливо).

Литеральные значения

В предыдущем примере мы столкнулись с двумя литеральными значениями: целым числом и строкой. Переменные других типов также обладают определенными литеральными значениями, которые сведены в приведенную ниже таблицу. Многие из них предполагают использование **суффиксов**, т. е. добавление в конец литерального значения некоторой последовательности символов, которая позволяет определить необходимый тип. Некоторые литеральные значения могут иметь несколько типов, которые определяются в время работы компилятора по контексту:

Тип(ы)	Значение	Суффикс	Пример/Допустимые значения
`bool`	Логическое	Отсутствует	`true` или `false`
`int, uint, long, ulong`	Целое	Отсутствует	`100`
`uint, ulong`	Целое	`u` или `U`	`100U`
`long, ulong`	Целое	`l` или `L`	`100L`
`ulong`	Целое	`ul, uL, Ul, UL, lu, lU, Lu` или `LU`	`100UL`
`float`	Вещественное	`f` или `F`	`1,5F`
`double`	Вещественное	Отсутствует, `d` или `D`	`1,5`
`decimal`	Вещественное	`m` или `M`	`1,5M`
`char`	Символьное	Отсутствует	`'a'` или escape-последовательность
`string`	Строковое	Отсутствует	`"a...a"`, может включать escape-последовательности

Литеральные строки

В этой главе нам уже встретилось несколько escape-последовательностей, которые можно использовать в литеральных строках. Представляется разумным привести здесь таблицу всех возможных escape-последовательностей для справочных целей.

Escape-последовательность	Выводимый символ	Код символа в кодировке Unicode
\'	Одиночная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\	Обратный слэш	0x005C
\0	Null	0x0000
\a	Тревога (выдает звуковой сигнал)	0x0007
\b	Удаление предыдущего символа (backspace)	0x0008
\f	Формирование строки (form feed)	0x000C
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Символ горизонтальной табуляции	0x0009
\v	Символ вертикальной табуляции	0x000B

В этой таблице последний столбец содержит шестнадцатеричные значения кодов символов в кодировке Unicode.

Кроме приведенных выше символов, с помощью escape-последовательности можно задать любой символ в кодировке Unicode. Такая последовательность должна состоять из стандартного символа \, за которым следует символ u и четырехзначное шестнадцатеричное значение (например, те четыре цифры, которые расположены после символа x в последнем столбце вышеприведенной таблицы).

Отсюда следует, что, например, следующие две строки эквивалентны:

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

Очевидно, что возможность использовать escape-последовательности в кодировке Unicode дает дополнительную гибкость.

Мы также можем применять "дословные" строки (verbatim). Это означает, что в строку включаются все символы, находящиеся между двумя двойными кавычками, в том числе символы конца строки и символы, для которых в противном случае потребовалось бы использование escape-последовательностей. Единственным исключением из этого правила является escape-последовательность для символа двойной кавычки, которая должна использоваться во избежание преждевременного завершения обработки строки. Чтобы получить дословную строку, необходимо поместить в ее начало символ @:

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

Эта строка может быть задана и обычным способом, а вот для следующего примера такой способ оказывается единственно возможным:

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

Дословные строки оказываются полезными, в частности, для имен файлов, поскольку в них обычно присутствует большое количество обратных слэшей. Использование обычных строк привело бы к необходимости использования двойных обратных слэшей во всей строке. Например:

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

Используя дословные литеральные строки, мы можем сделать запись более удобочитаемой. Следующая дословная строка эквивалентна вышеприведенной:

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

Как вы увидите дальше, строки представляют собой ссылочные типы, в отличие от всех остальных типов, с которыми вы познакомились в данной главе и которые являются значимыми типами. Одним из следствий этого является возможность присвоить строковой переменной значение Null: это будет означать, что данная строковая переменная не ссылается ни на какую строку. Подробнее этот вопрос будет рассматриваться ниже.

Объявление переменных и присваивание им значений

Для быстрого повторения давайте вспомним, что мы объявляли переменные, просто указывая их тип и имя, например:

```
int age;
```

Далее мы присваивали переменным значения с помощью оператора присванвания:

```
age = 25;
```

Запомните, что, перед тем как начать использовать переменную, вы должны ее инициализировать. В качестве такой инициализации может использоваться приведенное выше присваивание.

Существуют еще два способа, которыми мы можем воспользоваться и которые, скорее всего, встретятся вам в кодах на C#. Первый из них — это возможность одновременного объявления сразу нескольких переменных одного типа, чего можно добиться перечислением их имен через запятую после указания типа:

```
int xSize, ySize;
```

В данном случае обе переменные — `xSize` и `ySize` — описаны как целые числа.

Второй способ, который вам наверняка встретится, заключается в одновременном объявлении переменной и присванвании ей значения, что фактически означает объединение двух строк программы:

```
int age = 25;
```

Существует возможность совместного использования этих двух способов:

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

В данном случае соответствующие значения присваиваются и переменной `xSize`, и переменной `ySize`.

Обратите внимание, что следующий оператор:

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

приведет к инициализации только переменной `ySize`, переменная же `xSize` будет просто объявлена.

Выражения

Теперь, когда вы познакомились с объявлением и инициализацией переменных, настало время узнать, какие действия можно выполнять над ними. Для этих целей язык C# имеет целый набор операторов, в частности, оператор присваивания =, который мы уже использовали. Записывая операторы в определенном сочетании с переменными и литеральными значениями (при использовании совместно с операторами они называются **операндами**), мы можем создавать **выражения**, которые являются основными строительными блоками компьютерных программ.

Существуют различные операторы, начиная от самых простых и до чрезвычайно сложных, которые вряд ли вам встретятся где-либо, кроме как в математических приложениях. Простые операторы включают в себя все основные математические действия (например, оператор + производит сложение двух операндов), а сложные операторы предназначены для выполнения манипуляций над двоичным представлением содержимого переменной. Существуют также логические операторы, работающие с логическими значениями, и операторы присваивания типа =.

В данной главе основное внимание мы будем уделять математическим операторам и оператору присваивания, отложив логические операторы до следующей главы, в которой приступим к изучению булевой логики в контексте управления ходом выполнения программы.

Все операторы могут быть условно разделены на три категории:

- **Унарные** операторы, которые выполняют действие над единственным операндом.
- **Бинарные** операторы, которые выполняют действие над двумя операндами.
- **Тринарные** операторы, которые выполняют действие над тремя операндами.

Большинство операторов попадает в категорию бинарных; существует также несколько унарных операторов и единственный тринарный, называемый **условным** оператором (условный оператор представляет собой логический оператор, т. е. возвращает логическое значение; мы познакомимся с ним в следующей главе).

Математические операторы

Существует пять простых математических операторов, причем два из них имеют как бинарную, так и унарную форму. В приведенной ниже таблице перечислены все математические операторы, даны элементарные примеры их использования и показаны результаты их работы с простыми числовыми типами (целыми числами и числами с плавающей запятой):

Оператор	Значение	Пример использования в выражении	Результат
+	Бинарное	`var1 = var2 + var3;`	Переменной `var1` присваивается значение, которое представляет собой сумму значений переменных `var2` и `var3`
-	Бинарное	`var1 = var2 – var3;`	Переменной `var1` присваивается значение, получающееся в результате вычитания значения переменной `var3` из значения переменной `var2`

Продолжение таблицы

Оператор	Значение	Пример использования в выражении	Результат
*	Бинарное	`var1 = var2 * var3;`	Переменной `var1` присваивается значение, которое представляет собой результат перемножения переменных `var2` и `var3`
/	Бинарное	`var1 = var2 / var3;`	Переменной `var1` присваивается значение, которое представляет собой результат деления переменной `var2` на переменную `var3`
%	Бинарное	`var1 = var2 % var3;`	Переменной `var1` присваивается значение, которое представляет собой остаток от деления переменной `var2` на переменную `var3`
+	Унарное	`var1 = +var2;`	Переменной `var1` присваивается значение переменной `var2`
-	Унарное	`var1 = -var2;`	Переменной var1 присваивается значение переменной `var2`, умноженное на −1

Здесь приведены примеры с использованием простых численных типов, так как результаты, получаемые при действиях с другими простыми типами, могут оказаться неясными. Что бы вы хотели получить при сложении двух логических переменных? На самом деле мы не получим ничего, поскольку компилятор не позволит использовать оператор + (а также любые другие математические операторы) с переменными типа `bool`. Сложение переменных типа char также выглядит несколько запутанным. Если вы помните, переменные типа `char` на самом деле хранятся в виде чисел, поэтому результатом сложения двух переменных типа `char` также будет число (типа `int`, если быть точным). Такое действие представляет собой пример **неявного преобразования** типов, о чем мы будем говорить более подробно; некоторое внимание мы также уделим **явному преобразованию** типов, поскольку оно применимо в тех случаях, когда переменные `var1`, `var2` и `var3` относятся к разным типам.

Необходимо отметить, что бинарный оператор + имеет смысл и тогда, когда он используется с переменными типа `string`. В этом случае предыдущая таблица должна быть дополнена следующим:

Оператор	Значение	Пример использования в выражении	Результат
+	Бинарное	`var1 = var2 + var3;`	Переменной `var1` присваивается значение, которое представляет собой результат объединения двух строк, хранящихся в переменных `var2` и `var3`

Никакие другие математические операторы к строкам применяться не могут.

Два других оператора, которые нам необходимо рассмотреть, это операторы увеличения (инкремент) и уменьшения (декремент), каждый из которых является унарным оператором и может использоваться одним из двух способов: либо непосредственно перед операндом, либо сразу после него. Давайте посмотрим на результаты, которые получаются в случае простых выражений, а затем их обсудим.

Оператор	Значение	Пример использования в выражении	Результат
`++`	Унарное	`var1 = ++var2;`	Переменной `var1` присваивается значение переменной `var2+1`. Переменная `var2` увеличивается на единицу
`--`	Унарное	`var1 = --var2;`	Переменной `var1` присваивается значение переменной `var2-1`. Переменная `var2` уменьшается на единицу
`++`	Унарное	`var1 = var2++;`	Переменной `var1` присваивается значение переменной `var2`. Переменная `var2` увеличивается на единицу
`--`	Унарное	`var1 = var2--;`	Переменной `var1` присваивается значение переменной `var2`. Переменная `var2` уменьшается на единицу

Эти операторы всегда приводят к изменению значения, хранящегося в соответствующем операнде. Оператор ++ всегда увеличивает значение своего операнда на 1, а оператор -- уменьшает его на 1. Отличие заключается в том результате, который записывается в переменную `var1`, поскольку местоположение оператора определяет порядок выполнения действий. Когда любой из этих двух операторов помещен перед операндом, он выполняется прежде каких-либо других вычислений. Размещение любого из этих двух операторов после операнда приводит к тому, что оператор выполняется после завершения всех остальных вычислений для данного выражения.

Все сказанное заслуживает еще одного примера. Рассмотрим следующий код:

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

Вопрос: какое значение будет присвоено переменной `var1`? Перед тем как выражение начнет вычисляться, будет выполнен оператор --, расположенный перед переменной `var3`, что приведет к измененню ее значения с 6 на 5. Мы можем проигнорировать оператор ++, расположенный за переменной `var2`, поскольку он не будет оказывать никакого влияния до тех пор, пока вычисление этого выражения не будет завершено, поэтому в `var1` будет записан результат умножения 5 на 5, т. е. 25.

Эти простые унарные операторы оказываются чрезвычайно удобными в очень многих ситуациях. На самом деле они представляют собой просто сокращенную запись выражений вроде:

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

Оказывается, что такие выражения имеют очень широкое распространение, в частности, при организации циклов, с чем мы столкнемся в следующей главе.

Итак, теперь давайте рассмотрим, каким образом можно использовать математические операторы, а заодно познакомимся с еще двумя полезными понятиями.

Практикум: выполнение действий над переменными с помощью математических операций

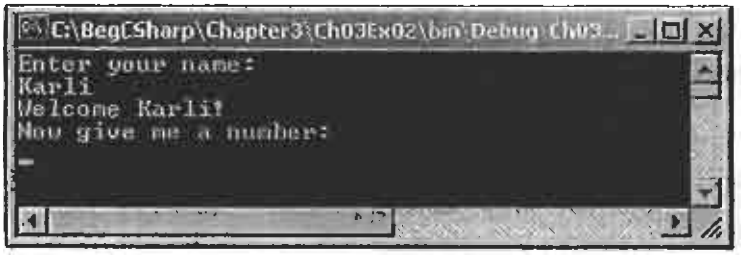
1. Создайте новое консольное приложение с именем `Ch03Ex02` в директории `C:\BegCSharp\Chapter3`.

2. Добавьте следующий код в `Class1.cs`:

```
static void Main(string[] args)
{
    double firstNumber, secondNumber;
    string userName;
    Console.WriteLine("Enter your name:");
    userName = Console.ReadLine();
    Console.WriteLine("Welcome {0}!", userName);
    Console.WriteLine("Now give me a number:");
        firstNumber = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Now give me another number:");
        secondNumber = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
        secondNumber, firstNumber + secondNumber);
    Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
        secondNumber, firstNumber, firstNumber - secondNumber);
    Console.WriteLine("The product of {0} and {1} is {2}.",
        firstNumber, secondNumber, firstNumber * secondNumber);
    Console.WriteLine("The result of dividing {0} by {1} is {2}.",
        firstNumber, secondNumber, firstNumber / secondNumber);
    Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
        firstNumber, secondNumber, firstNumber % secondNumber);
}
```

3. Выполните программу (см. рис. справа).





4. Введите свое имя и нажмите клавишу *Enter* (см. рис. слева).

5. Введите число, нажмите *Enter*, затем введите еще одно число и еще раз нажмите *Enter*:



Как это работает

Эта программа не только демонстрирует использование математических операторов, но в ней также появились два важных понятия, с которыми мы будем постоянно встречаться в учебных примерах:

- Ввод данных пользователем
- Преобразование типов

Для ввода данных, осуществляемого пользователем, применяется синтаксис, аналогичный синтаксису команды `Console.WriteLine()`, с которой вам уже приходилось встречаться. В данном случае используется команда `Console.ReadLine()`. Эта команда осуществляет вывод приглашения для ввода, который помещается в переменную типа `string`:

```
string userName;
Console.WriteLine("Enter your name:");
userName = Console.ReadLine();
Console.WriteLine("Welcome {0}!", userName);
```

Этот код выводит содержимое переменной `userName` на экран.

В данном примере мы также осуществляем считывание двух вводимых чисел. Здесь все оказывается несколько сложнее, так как команда `Console.ReadLine()` генерирует строку, а нам требуются числа. Тут вводится понятие **преобразования типов**. Более подробно оно будет описываться в главе 5, однако для начала давайте рассмотрим код, использованный в примере.

Во-первых, мы объявляем две переменные, в которых собираемся хранить вводимые числа:

```
double firstNumber, secondNumber;
```

Затем мы задаем приглашение и применяем с помощью `Console.ReadLine()` к полученной строке команду `Convert.ToDouble()`, позволяющую преобразовывать строки в тип `double`. Полученное значение присваивается первой из объявленных нами переменных — `firstNumber`:

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

Такой синтаксис является очень простым, поэтому вас не должно удивлять, что многие другие типы преобразований выполняются аналогичным образом.

Далее в программе точно так же происходит получение второго числа:

```
Console.WriteLine("Now give me another number:");
secondNumber = Convert.ToDouble(Console.ReadLine());
```

После этого выводятся результаты сложения, вычитания, умножения и деления двух введенных чисел, а также остаток от деления этих двух чисел, полученный с помощью оператора "остаток от деления" (%):

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
            secondNumber, firstNumber + secondNumber);
Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
            secondNumber, firstNumber, firstNumber - secondNumber);
Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
            secondNumber, firstNumber * secondNumber);
Console.WriteLine("The result of dividing {0} by {1} is {2}.", firstNumber,
            secondNumber, firstNumber / secondNumber);
Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
            firstNumber, secondNumber, firstNumber % secondNumber);
```

Обратите внимание, что в качестве параметров оператора `Console.WriteLine()` мы используем выражения вроде `firstNumber + secondNumber`, избегая стадию присваивания значения этих выражений промежуточным переменным:

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

Использование подобного синтаксиса позволяет делать код удобочитаемым и сокращает количество строк.

Операторы присваивания

До настоящего момента мы пользовалнсь простым оператором присваивания =, и кому-то может показаться удивительным, что существуют какие-либо другие операторы присваивания. Однако они действительно существуют и, что кажется еще более удивительным, оказываются весьма полезными!

Все другие операторы присваивания работают схожим с оператором = образом. Так же, как и у оператора =, результат их выполнения — присваивание расположенной слева переменной значения, вычисленного посредством операторов и операндов, расположенных в правой части.

Представим эти операторы и объяснение их работы в табличной форме:

Оператор	Значение	Пример использования в выражении	Результат
`=`	Бинарное	`var1 = var2;`	Переменной `var1` присваивается значение переменной `var2`
`+=`	Бинарное	`var1 += var2;`	Переменной `var1` присваивается значение, которое представляет собой сумму переменных `var1` и `var2`
`-=`	Бинарное	`var1 -= var2;`	Переменной `var1` присваивается значение, которое представляет собой разность, полученную в результате вычитания переменной `var2` из переменной `var1`
`*=`	Бинарное	`var1 *= var2;`	Переменной `var1` присваивается значение, которое представляет собой произведение переменных `var1` и `var2`
`/=`	Бинарное	`var1 /= var2;`	Переменной `var1` присваивается значение, полученное в результате деления переменной `var1` на переменную `var2`
`%=`	Бинарное	`var1 %= var2;`	Переменной `var1` присваивается значение, являющееся остатком от деления переменной `var1` на переменную `var2`

Как вы могли заметить, результатом использования дополнительных операторов присваивания является вовлечение переменной `var1` в вычисления, т. е. код вроде:

```
var1 += var2;
```

приводит к получению точно такого же результата, что и код:

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

Обратите внимание, что оператор += может использоваться и со строками, так же, как и оператор +.

Использование этих операторов, особенно при работе с длинными именами переменных, позволяет сделать программу намного более удобочитаемой.

Старшинство операторов

При вычислении выражения все операторы выполняются в определенной последовательности. Это не означает, что они всегда выполняются слева направо.

В качестве самого простого примера рассмотрим следующий оператор:

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

Здесь оператор + выполняется перед оператором =.

Существуют и другие ситуации, когда старшинство операторов не совсем очевидно, например:

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

В этом выражении сначала будет выполнен оператор *, затем оператор + и последним — оператор =. Это принятый в математике стандартный порядок выполнения операций, он приведет к получению того же самого результата, который можно ожидать при выполнении аналогичных арифметических вычислений на бумаге.

Мы также можем управлять порядком выполнения операторов с помощью скобок, например:

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

В данном случае в первую очередь будет выполняться то, что содержится в скобках, то есть оператор + будет выполнен перед оператором *.

Для всех рассматриваемых операторов порядок выполнения показан ниже; при этом операторы с одинаковым старшинством (такие как * и /) выполняются в порядке своего следования слева направо:

Старшинство	Операторы
Высшее	++, -- (используемые в качестве префиксов); +, - (унарные)
	*, /, %
	+, -
	=, *=, /=, %=, +=, -=
Низшее	++, -- (используемые в качестве суффиксов)

Для изменения порядка выполнения операторов используются скобки, как говорилось выше.

Пространства имен

Прежде чем мы двинемся дальше, представляется целесообразным потратить некоторое время на еще один важный аспект — **пространства имен**. Это способ, используемый в .NET, который позволяет создавать контейнеры для кода приложений, чтобы и код, и его составные части были однозначно идентифицированы.

Пространства имен используются также в качестве средства категоризации объектов в .NET Framework. Большинство таких объектов представляют собой определения типов, например, определения простых типов, которые подробно рассматриваются в данной главе (`System.Int32` и т. д.).

Код C# по умолчанию содержится в **глобальном пространстве имен**. Это означает, что к объектам в коде C# можно обратиться из любого другого кода в глобальном пространстве имен просто по их имени. Можно воспользоваться ключевым словом `namespace` для того, чтобы явно задать пространство имен для любого блока кода, заключенного в фигурные скобки. Имена, находящиеся в таком пространстве имен, если к ним обращаются не из данного пространства имен, должны **квалифицироваться**.

Квалифицированным именем называется имя, в котором содержится вся информация касательно его иерархии. Это означает, что если у нас имеется код, находящийся в одном пространстве имен, и необходимо воспользоваться именем, определенным в другом пространстве имен, то нам следует использовать ссылку на это пространство имен. В квалифицированных именах для разделения уровней пространств имен используется символ точки.

Например:

```
namespace LevelOne
{
   // программа, находящаяся в пространстве имен LevelOne

   // в ней описывается имя "NameOne"
}

// программа, находящаяся в глобальном пространстве имен
```

В этой программе описывается единственное пространство имен — `LevelOne`. (Обратите внимание, что в данном случае в программе не содержится никакого исполняемого кода. Это сделано для того, чтобы придать обсуждению максимально общий характер; напротив, в код помещен комментарий, в котором содержится это описание.) Код, содержащийся внутри пространства имен `LevelOne`, может просто ссылаться на имя `NameOne`, и никакой классификации не требуется. Однако в коде, находящемся в глобальном пространстве имен, необходимо использовать классифицированное имя `LevelOne.NameOne` для того, чтобы на него сослаться.

Внутри любого пространства имен мы можем описывать вложенные пространства имен, используя то же самое ключевое слово `namespace`. При обращение к вложенным пространствам имен следует указывать всю их иерархию, отделяя один уровень иерархии от другого с помощью точки. Это лучше всего проиллюстрировать с помощью примера. Рассмотрим следующие пространства имен:

```
namespace LevelOne
{
   // программа, находящаяся в пространстве имен LevelOne

   namespace LevelTwo
   {
      // программа, находящаяся в пространстве имен LevelOne.LevelTwo

      // в ней описывается имя "NameTwo"
   }
}

// программа, находящаяся в глобальном пространстве имен
```

В данном случае обращение к имени NameTwo из глобального пространства имен должно иметь вид LevelOne.LevelTwo.NameTwo, из пространства имен LevelOne — LevelTwo.NameTwo, а из пространства имен LevelOne.LevelTwo — NameTwo.

Здесь нужно отметить, что имена идентифицируются пространствами имен уникальным образом. Мы можем описать одно и то же имя "NameThree" как в пространстве имен LevelOne, так и в пространстве имен LevelTwo:

```
namespace LevelOne
{
   // здесь описывается имя "NameThree"

   namespace LevelTwo
   {
      // здесь описывается имя "NameThree"
   }
}
```

В данном случае описаны два разных имени, которые могут использоваться независимо друг от друга — LevelOne.NameThree и LevelOne.LevelTwo.NameThree.

После того, как пространство имен определено, появляется возможность использовать оператор using для упрощения доступа к содержащимся в нем именам. Оператор using как бы говорит: "Да, нам действительно понадобится доступ к именам из данного пространства имен, поэтому не заставляйте меня классифицировать их каждый раз". Например, в следующей программе предполагается, что код, находящийся в пространстве имен LevelOne, должен иметь доступ к именам пространства имен LevelOne.LevelTwo без какой бы то ни было классификации:

```
namespace LevelOne
{
   using LevelTwo;

   namespace LevelTwo
   {
      // здесь описывается имя "NameTwo"
   }
}
```

Код, находящийся в пространстве имен LevelOne, теперь может обращаться к LevelTwo.NameTwo просто как к NameTwo.

Бывают случаи, как в примере с NameThree, когда такой подход может приводить к конфликту между именами, находящимися в различных пространствах имен (при этом код, скорее всего, компилироваться не будет). В подобных случаях мы можем задать для пространства имен подставное имя (alias) в операторе using:

```
namespace LevelOne
{
   using LT = LevelTwo;

   // здесь описывается имя "NameThree"

   namespace LevelTwo
   {
      // здесь описывается имя "NameThree"
   }
}
```

Тогда в пространстве имен LevelOne мы сможем обращаться к LevelOne.NameThree просто как к NameThree, а к LevelOne.LevelTwo.NameThree как к LT.NameThree.

Операторы using оказывают влияние на то пространство имен, в котором они находятся, а также на все вложенные пространства имен, которые могут содержаться в данном пространстве имен. В вышеприведенном коде в глобальном пространстве имен использовать LT.NameThree нельзя. Однако если оператор using расположить следующим образом:

```
using LT = LevelOne.LevelTwo;

namespace LevelOne
{
   // здесь описывается имя "NameThree"
   namespace LevelTwo
   {
      // здесь описывается имя "NameThree"
   }
}
```

то использовать имя "LT.NameThree" можно будет и из глобального пространства имен, и из пространства имен LevelOne.

Существует еще один немаловажный момент, на который следует обратить внимание. Оператор using сам по себе не обеспечивает доступа к именам, находящимся в других пространствах имен. До тех пор, пока код из пространства имен не будет каким-либо способом привязан к нашему проекту (например, описан в исходном файле проекта или описан в каком-либо коде), привязанному к этому проекту, мы не получим доступа к содержащимся в нем именам. Более того, если код, в котором содержится некое пространство имен, привязан к нашему проекту, то мы обладаем доступом к содержащимся в нем именам независимо от использования оператора using. Оператор using всего лишь упрощает обращение к этим именам и позволяет сократить сильно удлиняющийся в противном случае код, делая его более понятным.

Вернувшись к программе ConsoleApplication1, с которой мы работали в самом начале данной главы, мы сможем обнаружить там следующие строки кода, относящиеся к пространствам имен:

```
using System;

namespace ConsoleApplication1
{
   . . .
}
```

В первой строке располагается оператор using, в котором объявляется, что пространство имен System будет использоваться во всей программе C# и доступ к нему из любых пространств имен данного файла должен осуществляться без использования классификации. Пространство имен System является корневым пространством имен в .NET Framework и включает все основные функциональные возможности, которые могут понадобиться для консольных приложений.

В следующей строке объявляется пространство имен для самого приложения ConsoleApplication1.

Итоги

В данной главе вы познакомились с основополагающей информацией, необходимой для создания работающих приложений на C#. Были рассмотрены основы синтаксиса C# и стандартный код консольного приложения, который генерируется VS при создании проекта консольного приложения.

Большая часть этой главы была посвящена вопросам использования переменных. В ней рассказывалось, что представляют собой переменные, каким образом их можно создавать, как присваивать им значения и как выполнять манипуляции над ними и над теми значениями, которые в них содержатся. Параллельно с этим рассматривались основные аспекты взаимодействия с пользователем: было продемонстрировано, каким образом консольное приложение может выводить текст и считывать информацию, вводимую пользователем. Для этого потребовалось выполнить преобразование типов, которое более подробно будет рассматриваться в главе 5.

Вы также научились составлять выражения из операторов и операндов и узнали, каким образом и в какой последовательности они выполняются.

И, наконец, в этой главе были рассмотрены пространства имен, которые имеют очень важное значение для темы данной книги. Хотя пока этот вопрос затронут несколько поверхностно, мы тем не менее заложили основу для дальнейшего обсуждения.

До настоящего момента выполнение всех наших программ происходило последовательно, строка за строкой. В следующей главе вы узнаете, как сделать программы более эффективными, управляя ходом их выполнения с помощью циклов и условных ветвлений.

Упражнения

1. Каким образом следует обращаться к имени `great` из программы, находящейся в пространстве имен `fabulous`?

```
namespace fabulous
{
   // программа в пространстве имен fabulous
}

namespace super
{
   namespace smashing
   {
      // здесь описывается имя great
   }
}
```

2. Какие из перечисленных ниже имен переменных недопустимы?

   a) `myVariableIsGood`

   b) `99Flake`

   c) `_floor`

   d) `time2GetJiggyWidIt`

   e) `wrox.com`

3. Является ли строка `"supercalifragilisticexpialidocious"` слишком длинной для того, чтобы быть присвоенной переменной типа `string`? Почему?

4. Перечислите (с учетом старшинства операторов) последовательность шагов, необходимых для 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*: следует вспомнить, что для преобразования вводимых чисел в тип `double` нами использовался оператор `Convert.ToDouble()`, а для преобразования типа `string` в тип `int` использовался оператор `Convert.ToInt32()`.



Управление порядком выполнения

У всех программ на C#, которые рассматривались до сих пор, имеется одна общая черта. Во всех случаях выполнение кода происходило последовательно сверху вниз, строка за строкой, без каких-либо пропусков. Однако если бы приложения всегда выполнялись таким образом, то наши возможности были бы сильно ограничены.

В настоящей главе мы изучим два способа управления порядком выполнения программ, т. е. последовательностью выполняемых в программе на C# строк. Вот эти способы:

- Ветвление — выполнение кода обусловлено результатами предшествующих вычислений, например: "выполнить этот код только в том случае, если значение переменной `myVal` меньше 10".
- Использование циклов — повторяющееся выполнение одних и тех же операторов (определенное количество раз или до тех пор, пока не выполнится контрольное условие).

Оба способа предполагают использование **булевой логики**. В предыдущей главе вы познакомились с типом `bool`, однако практически не использовали его. В этой главе мы собираемся наверстать упущенное и начнем с рассмотрения булевой логики, чтобы потом использовать ее в сценариях управления порядком выполнения программ.

Булева логика

Тип `bool`, представленный в предыдущей главе, может принимать одно из двух значений: `true` (истина) или `false` (ложь). Этот тип очень часто применяется для записи результата выполнения какой-либо операции, с тем, чтобы мы могли выполнить какие-либо определяемые им действия. В частности, тип `bool` используется для хранения результатов **сравнения**.

> *В порядке исторического отступления необходимо вспомнить (и отдать ему дань уважения) английского математика Джорджа Буля (George Boole), работы которого, написанные в середине XIX века, послужили основой для создания булевой логики.*

В качестве примера давайте рассмотрим ситуацию, приведенную в начале этой главы, когда выполнять код нужно только при значении переменной myVal меньше 10. Для этого необходимо владеть некоторым способом, позволяющим определять, является ли утверждение "myVal меньше 10" правдой или ложью; т. е. необходимо получить результат логического сравнения.

Логическое сравнение требует применения логических операторов сравнения (также известных под именем операторов отношения), которые приведены в расположенной ниже таблице. В этой таблице переменная var1 во всех случаях имеет тип bool, а типы переменных var2 и var3 меняются.

Оператор	Тип	Пример выражения	Результат
==	Бинарный	`var1 = var2 == var3;`	Переменной var1 присваивается значение true в том случае, если переменная var2 равна переменной var3, и значение false в противном случае
!=	Бинарный	`var1 = var2 != var3;`	Переменной var1 присваивается значение true в том случае, если переменная var2 не равна переменной var3, и значение false в противном случае
<	Бинарный	`var1 = var2 < var3;`	Переменной var1 присваивается значение true в том случае, если переменная var2 меньше переменной var3, и значение false в противном случае
>	Бинарный	`var1 = var2 > var3;`	Переменной var1 присваивается значение true в том случае, если переменная var2 больше переменной var3, и значение false в противном случае
<=	Бинарный	`var1 = var2 <= var3;`	Переменной var1 присваивается значение true в том случае, если переменная var2 меньше или равна переменной var3, и значение false в противном случае
>=	Бинарный	`var1 = var2 >= var3;`	Переменной var1 присваивается значение true в том случае, если переменная var2 больше или равна переменной var3, и значение false в противном случае

В программе мы можем использовать эти операторы для численных значений следующим образом:

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

В результате выполнения этого кода переменной isLessThan10 будет присвоено значение true в том случае, если значение переменной myVal меньше 10, а в противном случае — значение false.

Операторы сравнения могут использоваться также и для переменных других типов, например, для строк:

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

В данном случае переменной isKarli будет присвоено значение true только при условии, что строка, содержащаяся в переменной myString, имеет вид "Karli".

Аналогичным образом можно поступать и с логическими переменными:

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

Однако в этом случае допускается использование только операторов == и !=.

> *Распространенной ошибкой является безосновательное предположение, что если* `val1 < val2` *есть ложь, то* `val1 > val2` *— истина. Если* `val1 == val2`*, оба предыдущих выражения будут ложными. Автор специально упоминает об этой частой ошибке, поскольку сам допускал ее в прошлом.*

Существуют и некоторые другие операторы, специально предназначенные для работы с логическими значениями:

Оператор	Тип	Пример выражения	Результат
!	Унарный	`var1 = ! var2;`	Переменной `var1` присваивается значение `true` в том случае, если переменная `var2` имеет значение `false`, и значение `false`, если значение `var2` — `true` (операция "логическое не" (NOT))
&	Бинарный	`var1 = var2 & var3;`	Переменной `var1` присваивается значение `true` в том случае, если обе переменные `var2` и `var3` имеют значение `true`, и значение `false` в противном случае (операция "логическое и" (AND))
|	Бинарный	`var1 = var2 | var3;`	Переменной `var1` присваивается значение `true` в том случае, если хотя бы одна из переменных — `var2` или `var3` (или обе) — имеет значение `true`, и значение `false` в противном случае (операция "логическое или" (OR))
^	Бинарный	`var1 = var2 ^ var3;`	Переменной `var1` присваивается значение `true` в том случае, если одна из переменных — `var2` или `var3`, но не обе одновременно — имеет значение `true`, и значение `false` в противном случае (операция "исключающее или" (XOR))

Теперь мы можем записать последний пример следующим образом:

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

Для операторов & и | существуют аналоги:

Оператор	Тип	Пример выражения	Результат
&&	Бинарный	`var1 = var2 && var3;`	Переменной `var1` присваивается значение `true` в том случае, если обе переменные `var2` и `var3` имеют значение `true`, и значение `false` в противном случае (операция "логическое и" (AND))
||	Бинарный	`var1 = var2 || var3;`	Переменной `var1` присваивается значение `true` в том случае, если одна из переменных, `var2` или `var3` (или обе), имеет значение `true`, и значение `false` в противном случае (операция "логическое или" (OR))

Результат выполнения этих операций в точности соответствует результату выполнения операций & и |, однако существует одно важное отличие; оно заключается в способе достижения результата, позволяющем в некоторых случаях ускорить операцию. Оба эти оператора рассматривают значение своего первого операнда (в приведенной выше таблице это `var2`) и в определенных случаях могут вообще обходиться без обработки второго операнда (в нашем примере `var3`).

Если значение первого операнда оператора && — `false`, то нет необходимости рассматривать значение второго операнда, поскольку результатом все равно будет значение `false`. Аналогично оператор || вернет значение `true`, если его первый операнд имеет значение `true`, независимо от значения второго операнда. При выполнении операторов & и | всегда происходит вычисление обоих операндов.

С учетом этого при использовании операторов && и || вместо & и | можно ожидать небольшого увеличения быстродействия. Выгода будет особенно ощутима в тех приложениях, где задействовано много таких операторов. Существует эмпирическое правило, согласно которому при наличии возможности *всегда* следует использовать операторы && и ||.

Операторы работы с битами

По прочтении предыдущего раздела может возникнуть вопрос, для чего вообще существуют операторы & и |. Ответ заключается в том, что эти операторы могут использоваться для выполнения операций над численными значениями. Фактически они имеют дело с хранящейся в переменной последовательностью битов, а не со значением переменной как таковым.

Давайте шаг за шагом рассмотрим работу этих операторов, начав с оператора &. Он действует следующим образом: каждый бит первого операнда сравнивается с находящимся в такой же позиции битом второго операнда, а бит, записывающийся в аналогичной позиции результата, определяется следующим образом (см. табл. слева).

Бит операнда 1	Бит операнда 2	Бит результата выполнения операции &
1	1	1
1	0	0
0	1	0
0	0	0

Аналогичные действия выполняются и в случае применения оператора |, только результат вычисляется несколько иначе (см. табл. справа).

Бит операнда 1	Бит операнда 2	Бит результата выполнения операции |
1	1	1
1	0	1
0	1	1
0	0	0

В качестве примера рассмотрим операцию, выполняемую следующим кодом:

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

Здесь необходимо принять во внимание двоичное представление операндов `op1` и `op2`, соответственно 100 и 101. Результат этой операции получается путем сравнения двоичных битов, находящихся в одинаковых позициях в каждом из этих логических представлений:

- Первому слева биту результата присваивается значение 1, если оба крайних левых бита `op1` и `op2` равны 1, и значение 0 в противном случае.
- Следующему биту результата присваивается значение 1, если оба следующих бита `op1` и `op2` равны 1, и значение 0 в противном случае.
- Так продолжается для всех остальных битов.

В данном примере крайние биты слева у обоих операндов `op1` и `op2` равны 1, поэтому крайний левый бит результата также будет равен 1. В следующей позиции оба бита равны 0, а в третьей позиции биты равны соответственно 1 и 0; таким образом, второй и третий биты результата будут равны 0. В качестве окончательного результата в двоичном представлении будет получено число 100, т. е. результат выполнения этой операции будет равен 4.

Все вышесказанное можно проиллюстрировать следующим образом (см. рис. слева).

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

Аналогичный процесс происходит при использовании оператора |, с тем лишь отличием, что получающийся в результате бит равняется 1 в том случае, если хотя бы один из соответствующих битов операндов равен 1 (см. рис. справа).

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

Бит операнда 1	Бит операнда 2	Бит результата выполнения операции ^
1	1	0
1	0	1
0	1	1
0	0	0

Точно таким же способом можно использовать и оператор ^. В этом случае в результате появляется 1, если только у одного из операндов, но не у обоих одновременно, в соответствующей позиции находится единица (см. табл. слева).

В языке C# допускается использование унарного оператора работы с битами ~, воздействие которого на операнд состоит в инвертировании (изменении) всех его битов таким образом, что все биты, которые были равны 0, получают значение 1, и наоборот (см. табл. справа).

Бит операнда	Бит результата выполнения операции ^
1	0
0	1

Такие побитовые операции оказываются весьма полезными в самых разных ситуациях, поскольку они предлагают простой метод использования отдельных битов переменной для хранения информации. Рассмотрим простое представление цвета, в котором используется три бита для указания наличия красной, зеленой и синей составляющих. У нас появляется возможность задавать каждый из этих битов независимо и изменить цвет, определяемый всеми тремя битами, на один из следующих вариантов (см. табл. слева).

Биты	Десятичное представление	Значение
000	0	Черный
100	4	Красный
010	2	Зеленый
001	1	Синий
101	5	Пурпурный
110	6	Желтый
011	3	Голубой
111	7	Белый

Предположим, что мы храним эти значения в переменной типа `int`. Начиная с черного цвета или, что то же самое, со значения переменной типа `int`, равного 0, можно выполнить различные операции. Например:

```
int myColor = 0;
bool containsRed;
myColor = myColor | 2;
   // Добавлен бит зеленого цвета, значение переменной myColor стало 010
myColor = myColor | 4;
   // Добавлен бит красного цвета, значение переменной myColor стало 110
containsRed = (myColor & 4) == 4;
   // Выполняется проверка на наличие бита, отвечающего за красный цвет
```

В результате выполнения последней строки кода значение переменной `containsRed` будет `true`, поскольку "красный" бит в переменной `myColor` равен 1.

Этот способ может оказаться весьма полезным в том случае, если выполняемые операции используются для одновременной проверки нескольких битов (например, для 32 в случае значений типа `int`). Однако существуют более эффективные способы хранения дополнительной информации в отдельных переменных, для чего используются переменные сложных типов. Их обсуждению посвящена следующая глава.

Помимо описанных выше четырех операторов для работы с битами следует рассмотреть еще два оператора. А именно:

Оператор	Тип	Пример выражения	Результат
>>	Бинарный	`var1 = var2 >> var3;`	Переменной `var1` присваивается значение, которое получается при сдвиге двоичного содержимого переменной `var2` вправо на число битов, равное значению переменной `var3`
<<	Бинарный	`var1 = var2 << var3;`	Переменной `var1` присваивается значение, которое получается при сдвиге двоичного содержимого переменной `var2` влево на число битов, равное значению переменной `var3`

Работу этих операторов, обычно называемых операторами побитового сдвига, лучше всего проиллюстрировать с помощью небольшого примера:

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

В этом примере переменной `var1` будет присвоено значение 40. Действительно, двоичное представление числа 10 — "1010"; после сдвига на две позиции влево оно превратится в "101000" — двоичное представление числа 40. То, что мы сделали, можно изобразить как операцию умножения. Каждый бит при сдвиге на одну позицию влево увеличивает свое значение в два раза, соответственно сдвиг на две позиции эквивалентен умножению на 4. И наоборот, при сдвиге всех битов вправо возникает эффект деления операнда на два, при этом любой целый остаток теряется:

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

В данном случае `var1` будет содержать значение 5, а в результате выполнения следующего примера ее значение станет равным 2:

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

Представляется маловероятным, что вам придется активно пользоваться этими операторами, однако знать об их существовании полезно. Их применяют главным образом для получения предельно оптимизированного кода, когда использование других математических операций из-за их большей затратности оказывается неприемлемым. По этой причине рассмотренные здесь операции часто задействуют, например, в драйверах устройств или системных кодах.

Логические операторы присваивания

Последние операторы, которые нам осталось обсудить в этом разделе,— это комбинация рассмотренных выше операторов с оператором присваивания. Они во многом сходны с математическими операторами присваивания, рассматривавшимися в предыдущей главе (`+=`, `*=` и т. д.). Логические операторы приведены в следующей таблице:

Оператор	Тип	Пример выражения	Результат
`&=`	Бинарный	`var1 &= var2;`	Переменной `var1` присваивается значение, являющееся результатом выполнения операции `var1 & var2`
`|=`	Бинарный	`var1 |= var2;`	Переменной `var1` присваивается значение, являющееся результатом выполнения операции `var1 | var2`
`^=`	Бинарный	`var1 ^= var2;`	Переменной `var1` присваивается значение, являющееся результатом выполнения операции `var1 ^ var2`

Эти операторы используются как с логическими, так и с численными значениями, точно так же, как и операторы &, | и ^.

> *Обратите внимание, что при выполнении операторов &= и |= применяются операторы & и |, а не && и ||, что приводит к издержкам, характерным для этих более простых операторов.*

Для операторов побитового сдвига также существуют соответствующие операторы присваивания:

Оператор	Тип	Пример выражения	Результат
`>>=`	Унарный	`var1 >>= var2;`	Переменной `var1` присваивается значение, которое получается в результате сдвига двоичного содержимого переменной `var1` вправо на число битов, равное значению переменной `var2`
`<<=`	Унарный	`var1 <<= var2;`	Переменной `var1` присваивается значение, которое получается в результате сдвига двоичного содержимого переменной `var1` влево на число битов, равное значению переменной `var2`

Практикум: использование логических операторов и операторов работы с битами

1. Создайте новое консольное приложение с именем `Ch04Ex01` в директории `C:\BegCSharp\Chapter4`.

2. Добавьте следующий код в `Class1.cs`:

```
static void Main(string[] args)
{
    Console.WriteLine("Enter an integer:");
    int myInt = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Integer less than 10? {0}", myInt < 10);
    Console.WriteLine("Integer between 0 and 5? {0}",
                      (0 <= myInt) && (myInt <= 5));
    Console.WriteLine("Bitwise AND of Integer and 10 = {0}", myInt & 10);
}
```

3. Запустите приложение; в ответ на приглашение введите целое число:



Как это работает

Первые две строки кода выводят приглашение к вводу целого числа, что реализуется знакомым нам методом:

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

Оператор `Convert.ToInt32` используется для того, чтобы получнть целое число из введенной текстовой строки. Он представляет собой еще одну команду из того же семейства, к которому относится `Convert.ToDouble()`.

Оставшиеся три строки кода выполняют различные операции над введенным числом и выводят их результат. Мы рассмотрим работу данного кода, исходя из предположения, что пользователь ввел число 6, как это изображено на рисунке.

Сначала выводится результат выполнения операции `myInt < 10`. Поскольку `myInt` равняется 6, т. е. меньше 10, то в результате получается значение `true`, которое мы и наблюдаем в выходном потоке. При значении `myInt`, равном или превосходящем 10, результатом будет `false`.

Вторым будет выведено значение, которое предполагает более сложные вычисления: `(0 <= myInt) && (myInt <= 5)`. Здесь проводятся две операции сравнения, которые позволяют установить, является ли значение `myInt` большим или равным 0, а также меньшим или равным 5, и логическая операция AND, выполняемая над результатами сравнений. При значении `myInt`, равном 6, `(0 <= myInt)` возвращает значение `true`, а `(myInt <= 5)` возвращает значение `false`. Для получения окончательного результата вычисляется выражение `(true) && (false)`, которое дает значение `false`, что и отображается на консоли.

Наконец, мы производим побитовую операцию AND над значением переменной `myInt`. В качестве второго операнда используется число 10, которое имеет двоичное представление 1010. Если значение `myInt` равняется 6, что в двоичном виде выглядит как 110, то в результате выполнения этой операции получается двоичное число 10, или 2 в десятичном представлении (см. рис. справа).

	0	1	1	0		6
&	1	0	1	0	&	10
	0	0	1	0		2

Старшинство операторов с дополнениями

Теперь, когда мы рассмотрели несколько новых операторов, возникает необходимость внести в таблицу старшинства операторов, приведенную в предыдущей главе, определенные изменения:

Старшинство	Операторы
Высшее	++, -- (используемые в качестве префиксов); (), +, - (унарные), !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&
	^
	|
	&&
	||
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
Низшее	++, -- (используемые в качестве суффиксов)

Нам пришлось ввести в эту таблицу несколько новых уровней, зато теперь она точно определяет, каким образом следует вычислять выражения, например:

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

в котором оператор && выполняется после операторов <= и >=.

Здесь необходимо отметить тот факт, что совершенно не возбраняется использовать скобки с целью сделать выражения более понятными. Компилятор в любом случае разберется, в каком порядке следует выполнять операторы, а вот нам, людям, свойственно забывать о таких вещах (а ведь может возникнуть необходимость внести изменения в заданный порядок). Записав вышеприведенное выражение следующим образом:

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

мы сумеем решить эту проблему, явно указав последовательность выполнения операций.

Оператор goto

C# позволяет помечать строки кода, а затем непосредственно переходить к их выполнению с помощью оператора `goto`. У этого оператора имеются свои преимущества и недостатки. Основное преимущество заключается в том, что это очень простой способ управлять очередностью выполнения кодов. Однако чрезмерное его использование может привести к трудному для понимания "лоскутному" коду, что является его главным недостатком.

Для лучшего понимания давайте рассмотрим, каким образом может использоваться этот оператор.

Оператор `goto` имеет вид:

```
goto <имяМетки>;
```

Метки задаются таким образом:

```
<имяМетки>:
```

В качестве примера разберем следующий код:

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

Его выполнение осуществляется следующим образом:

- Переменная `myInteger` объявляется как имеющая тип `int` и ей присваивается значение 5.
- Оператор `goto` прерывает нормальный ход выполнения кода и передает управление на строку с меткой `myLabel:`.
- Значение переменной `myInteger` выводится на консоль.

Строка кода, выделенная в приведенной ниже программе серым цветом, *никогда* не будет выполняться:

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

В самом деле, если вы попытаетесь использовать этот код в приложении, то обнаружите, что при попытке его откомпилировать появится предупреждение "Unreachable code detected" ("Обнаружен невыполняемый код") с номером соответствующей строки.

> *Операторы `goto` используются на практике, но они могут приводить к очень большой путанице.*

В качестве примера "лоскутного" кода, получающегося при использовании оператора `goto`, рассмотрим следующую программу:

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

```
        goto start;
        addVal:
        myInteger += 10;
        goto writeResult;
```

Это вполне допустимый код, но как трудно в нем разобраться! Вы можете самостоятельно попрактиковаться и попытаться понять, что здесь происходит. Одиако прежде чем прнступить к этому, попробуйте понять смысл данного кода, просто взглянув на него. Вы заслуживаете похвалы, если вам удалось это сделать.

Мы еще вернемся к этому оператору, поскольку он предназначен для совместного использования с некоторымн другнми конструкциямн, рассматрнваемыми в настоящей главе (хотя, по правде говоря, автор не является сторонником его использования).

Ветвление

Ветвленне — это способ управлять тем, какая строка кода должна выполняться следующей. Строка, на которую следует передать управление, определяется с помощью некоторого условного оператора. Работа такого условного оператора основана на выполненнн сравнення контрольного значения и одного или нескольких возможных значений с использованием булевой логики.

В этом разделе мы рассмотрим три способа ветвления, имеющнхся в C#:

- Тринарный оператор
- Оператор `if`
- Оператор `switch`

Тринарный оператор

Наиболее простой способ выполнения проверки — использование **тринарного** (или условного) оператора, упоминавшегося в предыдущей главе. Вы уже позиакомилнсь с унарными операторами, которые выполняются над одним операндом, и бинарными операторами, которые выполняются над двумя операндами, поэтому у вас вряд ли вызовет удивление тот факт, что данный оператор выполияется над тремя операндами. Снитаксис тринарного оператора имеет следующий вид:

```
<проверка> ? <результатЕслиИстина> : <результатЕслиЛожь>
```

В данном случае условие *<проверка>* вычисляется для получения логического значения, и в зависимости от него результатом выполиения этого оператора будет *<результатЕслиИстина>* или *<результатЕслиЛожь>*.

Мы можем использовать этот оператор следующим образом:

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

Результатом выполиения тринариого оператора будет одна из двух строк, каждая из которых может быть присвоена переменной `resultString`. Выбор того, какую именно из этих двух строк следует присвоить, производится путем сравиения значения переменной `myInteger` с 10, причем если значение переменной меньше 10, присванвается первая строка, а если больше или равно 10 — вторая. Например, в том случае, если значение переменной `myInteger` равняется 4, переменной `resultString` будет присвоена строка `"Меньше 10"`.

Описанный оператор вполне подходит для выполнения простых присваиваний, как в приведенном примере, однако для выполнения больших объемов кода он не приспособлен. Здесь лучше использовать оператор `if`.

Оператор If

Оператор `if` является гораздо более гибким способом принятия решений. В отличие от операторов `?:`, операторы `if` не имеют результата (поэтому их нельзя использовать в операторах присваивания), зато появляется возможность применять их для выполнения каких-либо других операторов при соблюдении определенного условия.

Наипростейшей формой оператора `if` является следующая:

```
if (<условие>)
   <код, который выполняется, если вычисление <условия> дает результат true>;
```

Когда происходит вычисление *<условия>* (а в результате этих вычислений должно получаться логическое значение, иначе код не пройдет компиляцию), строка кода, расположенная под оператором, будет выполнятся только в случае значения `true`. После того как эта строка кода будет выполиена, а также если эта строка не будет выполняться (из-за того, что получено значение `false`), работа продолжится со следующей за условиым оператором строки кода.

Можно задать дополнительный код, воспользовавшись оператором `else` в сочетании с оператором `if`. Этот оператор будет выполняться только в том случае, если в результате вычисления *<условия>* будет получено значение `false`:

```
if (<условие>)
   <код, который выполняется, если вычисление <условия> дает результат true>;
else
   <код, который выполняется, если вычисление <условия> дает результат false>;
```

Каждый раздел кода может занимать несколько строк, для чего используются фигурные скобки:

```
if (<условие>)
{
   <код, который выполняется, если вычисление <условия> дает результат true>;
}
else
{
   <код, который выполняется, если вычисление <условия> дает результат false>;
}
```

В качестве простого примера перепишем код из предыдущего раздела, в котором использовался тринарный оператор, с помощью оператора `if`:

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

Поскольку результат выполнения оператора `if` нельзя присваивать переменным, то потребуется отдельный шаг:

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

Такой код, хотя он и многословнее, гораздо проще читать и понимать, чем тот, в котором использовался тринарный оператор; к тому же такой код оказывается более гибким.

Давайте обратимся к примеру.

Практикум: использование оператора if

1. Создайте новое консольное приложение с именем `Ch04Ex02` в директории `C:\BegCSharp\Chapter4`.

2. Добавьте следующий код в `Class1.cs`:

```
static void Main(string[] args)
{
   string comparison;
   Console.WriteLine("Enter a number:");
   double var1 = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("Enter another number:");
   double var2 = Convert.ToDouble(Console.ReadLine());
   if (var1 < var2)
      comparison = "less than";
   else
   {
      if (var1 == var2)
         comparison = "equal to";
      else
         comparison = "greater than";
   }
   Console.WriteLine("The first number is {0} the second number.", comparison);
}
```

3. Запустите код и введите два числа в ответ на приглашения (см. рис. справа).



Как это работает

Первая часть кода вам уже знакома: здесь происходит получение двух значений типа `double` из строк, введенных пользователем:

```
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 переменной `comparison` типа `string`, в зависимости от значений, полученных для `var1` и `var2`, присваивается строка:

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

Если данное условие не выполняется, то это значит, что `var1` либо больше, либо равно `var2`. Тогда в разделе `else` первого сравнения делается еще одно, вложенное сравнение:

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

Раздел `else` второго сравнения будет достигнут только в том случае, если `var1` больше, чем `var2`:

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

В заключение происходит вывод значения переменной `comparison` на консоль:

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

Использование вложенных операторов представляет собой лишь один из возможных путей решения этой задачи. С тем же успехом мы могли бы написать следующее:

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

Недостатком такого способа является необходимость выполнять все три сравнения независимо от значений переменных `var1` и `var2`. В первом же случае выполняется только одно сравнение, если `var1 < var2` есть `true`, и два сравнения в противном случае (мы также выполняем сравнение `var1 == var2`), т. е. выполняется меньшее число строк кода. Разница в быстродействии несущественна, однако она может иметь значение для тех приложений, для которых скорость выполнения является критичной.

Проверка большего количества условий с помощью оператора if

В предыдущем примере мы проверяли три условия для значения переменной `var1`, которые охватывали все возможные для нее значения. Однако в некоторых случаях может потребоваться выполнять проверку на какие-то конкретные значения, например, 1, 2, 3, 4 и т. д. Используя тот же подход, мы получим неудобоваримый вложенный код вроде:

```
if (var1 == 1)
{
   // выполнение каких-либо действий
}
else
{
   if (var1 == 2)
   {
      // выполнение каких-либо других действий
   }
   else
   {
      if (var1 == 3 || var1 == 4)
      {
         // выполнение каких-либо других действий
      }
      else
      {
         // выполнение каких-либо других действий
      }
   }
}
```

Существует очень распространенная ошибка, когда условие в третьем сравнении записывается как `if (var1 == 3 || 4)`. *В таком случае в полном соответствии со старшинством операторов сначала будет выполнен оператор* `==`, *в результате чего оператор* `||` *будет выполняться с операндами логического и численного типов. Это приведет к ошибке.*

В такой ситуации разумнее применить несколько иную систему отступов и объединения элементов кода для блоков `else` (т. е. использовать после блока `else` одну строку кода, а не целый блок кода). Если мы поступим таким образом, то получим следующую структуру, содержащую операторы `else if`:

```
if (var1 == 1)
{
   // выполнение каких-либо действий
}
else if (var1 == 2)
{
   // выполнение каких-либо других действий
}
else if (var1 == 3 || var1 == 4)
{
   // выполнение каких-либо других действий
}
else
{
   // выполнение каких-либо других действий
}
```

На самом деле операторы `else if` представляют собой два отдельных оператора, и вышеприведенный код идентичен первоначальному варианту. Однако такой код является более удобочитаемым.

Когда необходимо выполнять такое большое количество сравнений, как в этом примере, стоит рассмотреть альтернативную возможность организации ветвления.

Оператор switch

Оператор `switch` очень похож на оператор `if` в том смысле, что он также обеспечивает условное выполнение кода в зависимости от результата выполненной проверки. Однако оператор `switch` позволяет производить проверку для многих значений переменной за один проход вместо того, чтобы проверять всего лишь одно условие. Эти проверки ограничены отдельными значениями и не допускают конструкций типа "больше, чем X", поэтому их использование имеет некоторые отличия; однако такой способ является очень эффективным.

Принципиальная структура оператора `switch` имеет следующий вид:

```
switch (<контрольнаяПеременная>)
{
   case <значение1>:
      <код, который должен выполняться в том случае,
                        если <контрольнаяПеременная> == <значение1> >
      break;
   case <значение2>:
      <код, который должен выполняться в том случае,
                        если <контрольнаяПеременная> == <значение2> >
      break;
   ...
```

```
        case <значениеN>:
           <код, который должен выполняться в том случае,
                                  если <контрольнаяПеременная> == <значениеN> >
           break;
        default:
           <код, который должен выполняться в том случае,
                              если <контрольнаяПеременная> != ни одному из значений>
           break;
     }
```

Значение, которым обладает *<контрольнаяПеременная>*, сравнивается со всеми значениями *<значениеX>*, каждое из которых имеет свой собственный оператор `case`; в случае совпадения значений выполняется код, предусмотренный для данного условия. Если ни одно из значений не совпало, будет выполняться код из раздела `default` (при условии, что этот блок не пуст). После завершения выполнения кода в некотором разделе выполняется еще одна команда — `break`. В случае выполнения кода в одном из блоков `case` переход к выполнению другого оператора `case` не происходит.

> *Такая логика выполнения программы является одним из отличий языка C# от языка C++, в котором при обработке операторов `case` разрешается переходить от выполнения одного оператора к выполнению другого.*

Оператор `break` в данном случае просто прерывает выполнение оператора `switch`, и работа программы продолжается со следующего оператора.

В программе на C# имеются и другие способы избежать передачи управления от одного оператора `case` другому. Мы можем использовать либо оператор `return`, который приведет к прерыванию выполнения текущей функции, а не просто оператора `switch` (более подробно об этом будет рассказано в главе 6), либо оператор `goto`. Оператор `goto` (он описан выше) в данном случае будет работать, поскольку операторы `case` в программе на C# фактически выполняют роль меток. Например:

```
switch (<контрольнаяПеременная>)
{
   case <значение1>:
      <код, который должен выполняться в том случае,
                                  если <контрольнаяПеременная> == <значение1> >
      goto case <значение2>;
   case <значение2>:
      <код, который должен выполняться в том случае,
                                  если <контрольнаяПеременная> == <значение2> >
      break;
   . . .
```

Имеется единственное исключение из общего правила, гласящего, что после выполнения одного оператора `case` не может выполняться другой оператор `case`. Если мы разместим несколько операторов `case` подряд (объединим их) перед каким-либо блоком кода, мы фактически организуем одновременное выполнение проверки нескольких условий. Если хотя бы одно из этих условий окажется выполненным, то управление будет передано на этот блок кода. Например:

```
switch (<контрольнаяПеременная>)
{
   case <значение1>:
   case <значение2>:
      <код, который должен выполняться в том случае,
                                  если <контрольнаяПеременная> == <значение1>
                                  или <контрольнаяПеременная> == <значение2> >
      break;
   . . .
```

Обратите внимание, что эти условия также относятся и к оператору `default`. Не существует никакого правила, требующего, чтобы этот оператор обязательно находился в конце всего перечня сравнений, и при желании мы можем объединять его с операторами `case`. Включение оператора, прерывающего нормальное выполнение программы — `break`, `goto` или `return`, — гарантирует правильную последовательность выполняемых действий в любом случае.

Все переменные *<значениеХ>*, использующиеся для сравнений, должны иметь постоянное значение. Один из способом добиться этого — применение литеральных значений:

```
switch (myInteger)
{
   case 1:
      <код, который должен выполняться в том случае, если myInteger == 1 >
      break;
   case -1:
      <код, который должен выполняться в том случае, если myInteger == -1 >
      break;
   default:
      <код, который должен выполняться в том случае,
                                 если myInteger != ни одному из значений,
                                 использовавшихся для сравнения>
      break;
}
```

Другой способ получить постоянные значения переменных — это использование переменных-констант. Переменные-константы являются точно такими же переменными, как и все остальные, за исключением одного ключевого момента: значение, которое в них содержится, *не может быть изменено ни при каких условиях*. После того как мы присвоили переменной-константе какое-либо значение, оно будет оставаться неизменным на протяжении выполнения всей программы. Переменные-константы могут оказаться удобными в данном случае, поскольку зачастую код легче понимать, когда используемые для сравнения значения не видны в момент выполнения сравнения.

Мы можем объявить переменную-константу, добавив ключевое слово `const` к описанию типа переменной; при этом мы *обязаны* сразу определить ее значение. Например:

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

Этот код является вполне допустимым, однако если мы попытаемся выполнить следующий код:

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

то получим сообщение об ошибке на этапе компиляции. Также приведет к ошибке и попытка каким-либо образом изменить содержимое переменной-константы после первоначального присваивания.

Давайте рассмотрим пример оператора `switch`, в котором используются переменные-константы.

Практикум: использование оператора switch

1. Создайте новое консольное приложение с именем `Ch04Ex03` в директории `C:\BegCSharp\Chapter4`.

2. Добавьте следующий код в Class1.cs:

```
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?");
   name = Console.ReadLine();
   switch (name.ToLower())
   {
      case myName:
         Console.WriteLine("You have the same name as me!");
         break;
      case sexyName:
         Console.WriteLine("My, what a sexy name you have!");
         break;
      case sillyName:
         Console.WriteLine("That's a very silly name.");
         break;
   }
   Console.WriteLine("Hello {0}!", name);
}
```



3. Запустите приложение и введите имя (см. рис. слева).

Как это работает

В этом коде определяются три переменные-константы типа `string`, затем от пользователя принимается строка, после чего на консоль выводится текст в зависимости от введенной строки. В данном случае строки представляют имена.

Когда мы выполняем сравнение введенного имени (переменной `name`) со значениями наших констант, то мы прежде всего переводим его в нижний регистр посредством оператора `name.ToLower()`. Этот стандартный оператор, который может работать с любыми переменными типа `string`, оказывается очень удобным в тех случаях, когда нет уверенности в том, что именно ввел пользователь. Такой способ гарантирует, что все строки типа `"Karli"`, `"kArli"`, `"karli"` и т. д. совпадут с контрольной строкой `"karli"`.

Собственно оператор `switch` занимается поиском совпадения введенной строки с теми константами, которые мы определили в начале программы, и, в случае успеха, выводит персональное приветствие пользователю. Если совпадения не произошло, то мы просто приветствуем пользователя.

Операторы `switch` не имеют ограничений на количество содержащихся в них разделов `case`:, поэтому вы можете расширить этот код и охватить все имена, которые только могут придти вам в голову... но в этом случае вам придется изрядно поработать!

Организация циклов

Цикл — это участок кода, в котором операторы выполняются по несколько раз. Такой способ в ряде случаев бывает чрезвычайно удобным, поскольку позволяет повторять необходимые действия столько раз, сколько требуется (тысячи и даже миллионы), без необходимости выписывать соответствующий код такое же количество раз.

В качестве простого примера рассмотрим программу, вычисляющую количество денег, которое окажется на счете в банке по истечении десятилетнего срока, исходя из предположения, что проценты начисляются ежегодно и никакие суммы не снимаются и не добавляются на счет:

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

Выписывать один и тот же код 10 раз весьма расточительно, и потом, как быть, если потребуется изменить срок с 10 лет на другое значение? Придется вручную копировать строку с кодом необходимое число раз, что добавит немало головной боли!

К счастью, это не придется делать. Достаточно выполнить необходимые инструкции в цикле требуемое число раз.

Другой тип цикла — это цикл, вычисления в котором продолжаются до тех пор, пока не будет выполнено определенное условие. Такие циклы несколько проще (хотя не менее полезны), чем описанные выше, поэтому мы с них и начнем.

Цикл do

Цикл `do` работает следующим образом. Код, который помечен как выполняющийся в цикле, выполняется первый раз, затем проверяется некоторое логическое условие, и если результат равен `true`, то этот код выполняется еще раз, и т. д. Цикл заканчивается, когда условие перестает выполняться.

Цикл `do` имеет следующую структуру:

```
do
{
   <код, выполняющийся в цикле>
} while (<условие>);
```

где в результате вычисления *<условия>* получается логическое значение.

> *Точка с запятой после оператора `while` является обязательной; весьма распространена ошибка, когда ее забывают.*

Мы можем использовать этот цикл, например, для того, чтобы вывести в столбец числа от 1 до 10:

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

В данном случае мы использовали оператор `++` в виде суффикса, который увеличивает значение `i` на единицу после его вывода на экран; поэтому необходимо выполнять проверку `i <= 10`, чтобы число 10 попало в набор выводимых чисел.

Теперь давайте применим этот цикл в приведенной выше программе, определяющей накопления на банковском счете за 10 лет. С его помощью мы подсчитаем, сколько лет потребуется, чтобы на счету накопилась определенная сумма денег, при заданных начальной сумме и процентной ставке.

Практикум: использование циклов do

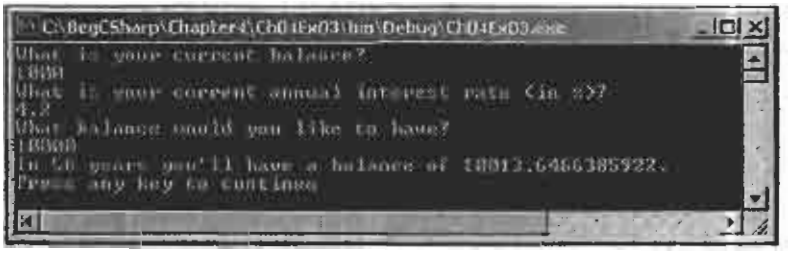
1. Создайте новое консольное приложение с именем `Ch04Ex04` в директории `C:\BegCSharp\Chapter4`.

2. Добавьте следующий код в `Class1.cs`:

```
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;
   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);
}
```

3. Запустите программу и введите какие-нибудь значения:



Как это работает

Эта программа повторяет вычисление ежегодного баланса с фиксированной процентной ставкой столько раз, сколько потребуется для того, чтобы баланс начал удовлетворять условию завершения. Мы используем счетчик лет, увеличивая его на единицу при каждом проходе по циклу:

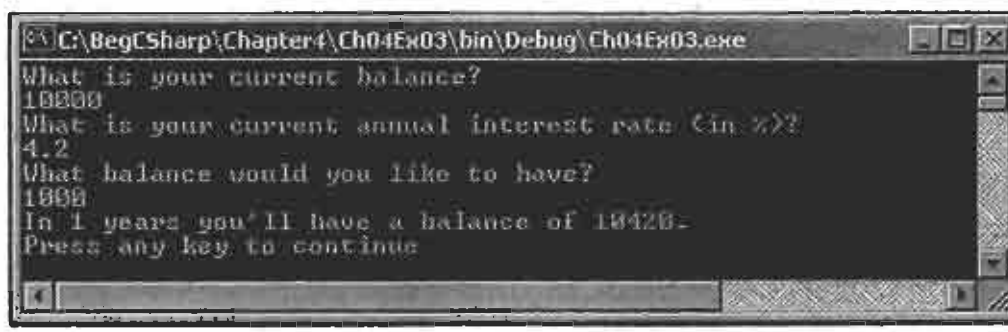
```
int totalYears = 0;
do
{
   balance *= interestRate;
   ++totalYears;
}
while (balance < targetBalance);
```

Затем мы используем значение переменной `totalYears` в качестве одной из составляющих выходного потока:

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

Такой способ использования оператора ?: (тринарного), пожалуй, является наиболее распространенным: он форматирует текст по условию с использованием минимального количества кода. Здесь мы применяем этот оператор для вывода символа "s" после слова "year" в том случае, если значение переменной `totalYears` не равно единице. (Буква "s" в английском языке служит для образования множественного числа имени существительного.— Прим. пер.)

К сожалению, эта программа не является оптимальной. Рассмотрим ситуацию, когда баланс, который требуется получить, оказывается меньше текущего баланса. Вот какой выходной поток получится в этом случае:



Цикл `do` *всегда* выполняется хотя бы один раз. Иногда, как в нашей ситуации, это не является правильным решением. Естественно, мы можем 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);
```

Однако это усложнение программы, которого можно избежать. Лучшим решением будет использование цикла `while`.

Цикл while

Цикл `while` очень похож на цикл `do`, однако у него имеется одно важное отличие: вычисление логического условия в цикле `while` происходит в начале цикла, а не в конце. Если в результате проверки условия будет получено значение `false`, то цикл `while` *вообще не будет выполняться* и управление передается коду, расположенному после этого цикла.

Цикл while оформляется следующим образом:

```
while (<условие>)
{
   <код, выполняющийся в цикле>
}
```

Он может использоваться практически так же, как и цикл do. Например:

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

Выполнение этого кода приведет к тому же результату: числа от 1 до 10 будут выведены в столбец.

Давайте изменим программу так, чтобы в ней использовался цикл while.

Практикум: использование цикла while

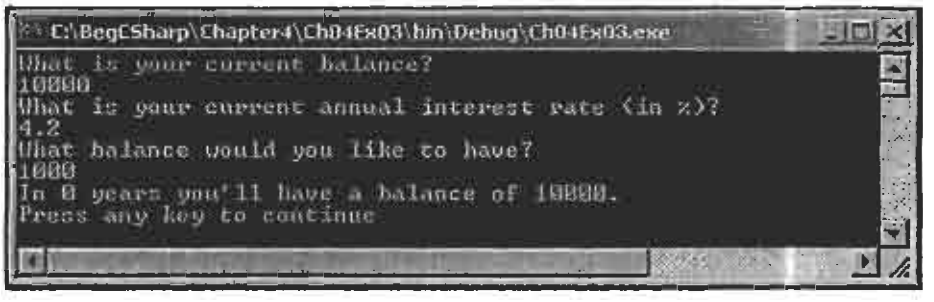
1. Откройте консольное приложение с именем Ch04Ex05 в директории C:\BegCSharp\Chapter4.

2. Модифицируйте код следующим образом (в качестве отправной точки используйте код из Ch04Ex04 и не забудьте удалить оператор while в конце исходного 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);
}
```

3. Запустите программу еще раз, однако теперь введите требуемый баланс меньше начального:



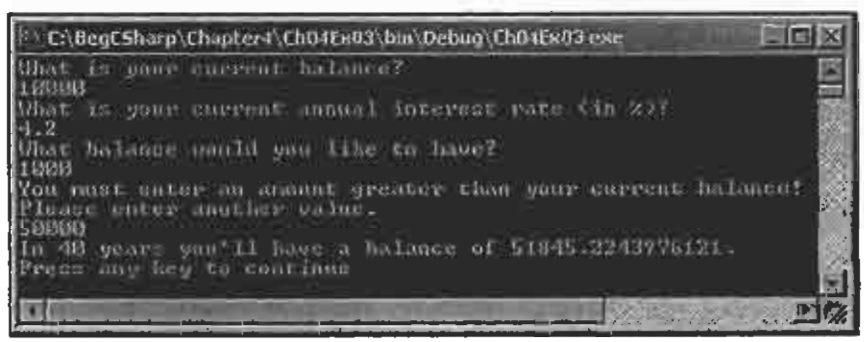
Как это работает

Простая замена цикла `do` циклом `while` позволила решить проблему, возникшую в предыдущем примере. Переместив проверку логического условия в начало цикла, мы предусмотрели ситуацию, при которой вообще не требуется выполнять цикл и можно сразу перейти к выводу результата.

Конечно, в данном случае существуют и альтернативные варианты. Например, можно осуществлять проверку вводимой пользователем информации, чтобы убедиться в превышении начальной суммы конечным балансом. Для этого поместим в цикл соответствующий код:

```
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.");
}
while (targetBalance <= balance);
```

Это позволит отвергнуть значения, лишенные смысла, и выходной поток будет выглядеть так:



Проверка допустимости информации, вводимой пользователем, является важным моментом при разработке приложений; по ходу книги вам встретится много таких примеров.

Цикл for

Последний тип цикла, который рассматривается в данной главе,— это цикл `for`. Он относится к тому типу циклов, которые выполняются заранее заданное количество раз и сами отвечают за организацию счетчика цикла. Для организации цикла `for` требуется следующая информация:

- Начальное значение для инициализации переменной цикла
- Условие для продолжения выполнения цикла, зависящее от переменной цикла
- Операция, которая будет выполняться над переменной цикла по завершении очередного прохода цикла

Например, если необходимо организовать цикл с переменной цикла, изменяющейся от 1 до 10 с шагом, равным единице, то в этом случае начальное значение

равно 1, условием для продолжения цикла будет "переменная цикла меньше или равна 10", а операцией, выполняющейся по окончании каждого прохода цикла, будет прибавление значению переменной цикла единицы. Эта информация должна быть размещена в структуре цикла `for` следующим образом:

```
for (<инициализация>; <условие>; <операция>)
{
   <код, выполняющийся в цикле>
}
```

Этот код работает так же, как и следующий цикл `while`:

```
<инициализация>;
while (<условие>)
{
   <код, выполняющийся в цикле>
   <операция>
}
```

Однако формат цикла `for` оказывается более понятным, поскольку все необходимые параметры, определяющие цикл, собраны воедино, а не распределены по разным операторам.

При рассмотрении циклов `do` и `while` мы приводили пример с выводом чисел от 1 до 10. Давайте взглянем на код, позволяющий сделать то же самое с помощью цикла `for`:

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

Целой переменной цикла с именем `i` присваивается начальное значение 1, после чего она увеличивается на 1 в конце каждого прохода. В теле цикла происходит вывод значения `i` на консоль.

Обратите внимание, что когда начинает выполняться код, расположенный за циклом, переменная `i` обладает значением, равным 11. Дело в том, что в конце прохода цикла, в котором переменная `i` равняется 10, ее значение увеличивается на единицу до проверки условия `i <= 10`. Очевидно, что после проверки выполнение цикла прекращается.

Как и циклы `while`, циклы `for` выполняются только в том случае, если проверка условия дает `true` перед первым проходом цикла, т. е. код, находящийся внутри цикла, может вообще ни разу не выполняться.

В заключение отметим, что у нас имеется возможность объявить переменную цикла в качестве составной части оператора `for`, для чего вышеприведенный пример следует переписать следующим образом:

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

В этом случае переменная `i` будет недоступна коду, находящемуся вне цикла (подробнее об этом см. главу 6, раздел "Область действия переменных").

Давайте рассмотрим пример использования цикла `for`. Поскольку мы уже достаточно хорошо знакомы с циклами, постараемся сделать его более интересным: в нем будет выводиться множество Мандельброта (Mandelbrot set) (правда, с помощью обыкновенных текстовых символов, что, конечно, выглядит не так захватывающе).

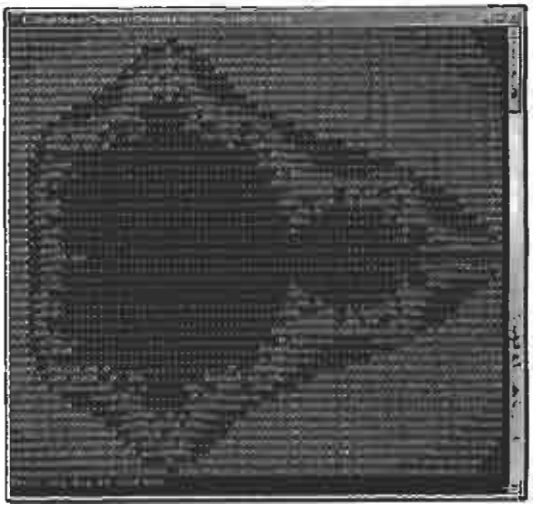
Практикум: использование цикла for

1. Создайте новое консольное приложение с именем `Ch04Ex06` в директории `C:\BegCSharp\Chapter4`.

2. Добавьте следующий код в `Class1.cs`:

```
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");
   }
}
```

3. Запустите программу (см. рис. справа).



Как это работает

Сейчас мы не будем подробно описывать, каким образом вычисляются множества Мандельброта, однако основные моменты необходимо объяснить для понимания ролн циклов в этой программе. Если вы не интересуетесь математикой, то можете смело пропустить следующне два абзаца, поскольку в данном случае самое важное — это понять, как устроена программа.

Каждая точка на рисунке Мандельброта соответствует комплексному числу вида `N = x + y*i`, где `x` — это действительная часть, `y` — мнимая часть, а `i` — корень квадратный из −1. Координаты `x` и `y` рисунка соответствуют действительной и мнимой частям комплексного числа.

Для каждой точки рисунка рассматривается аргумент `N`, который представляет собой корень квадратный из `x*x + y*y`. Если это значение больше или равно 2, то данная позиция имеет значение 0. Если же аргумент числа `N` меньше двух, то происходит замена `N` на `N*N - N` (что дает `N = (x*x - y*y - x) + (2*x*y - y)*i`) и снова выполняется проверка числа `N`. Если это значение оказывается большим или равным 2, позиция, соответствующая этому числу, будет иметь значение 1. Этот процесс продолжается до тех пор, пока данной точке рисунка не удастся присвоить некое число либо пока не будет превышено заранее заданное число итераций.

Основываясь на значениях, присвоенных каждой точке рисунка, в графической среде каждой точке на экране можно сопоставить пиксель определенного цвета. Однако поскольку мы используем алфавитно-цифровой дисплей, то будем просто выводить на экран различные символы.

Давайте рассмотрим код и содержащиеся в нем циклы.

В первую очередь мы объявили переменные, необходимые для дальнейших вычислений:

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

Здесь `realCoord` и `imagCoord` представляют действительную и комплексную части числа `N`, а остальные переменные типа `double` предназначены для хранения промежуточной информации, получающейся в процессе вычислений. В переменной `iterations` хранится количество итераций, потребовавшееся для того, чтобы аргумент `N` (`arg`) стал большим или равным 2.

Далее мы организовали два цикла `for`, которые позволяют пройти по всем координатам нашего изображения (в них для изменения счетчиков циклов используется более сложный, чем у операторов `++` и `--`, синтаксис):

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

Автор специально подобрал границы, которые позволяют продемонстрировать осиовную часть множества Мандельброта. Если у вас возникнет желание "увеличить" это изображение, то можете поэкспериментировать с границами сами.

Внутри этих двух циклов располагается код, относящийся к единственной точке множества Мандельброта и дающий то значение `N`, с которым ведется дальнейшая работа. Именно здесь вычисляется количество необходимых итераций и значение символа, выводимого в данной точке.

Сначала производится инициализация нескольких переменных:

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

Затем для осуществления итерационного процесса используется цикл `while`, а не цикл `do`, так как если начальное значение `N` больше 2, искомым ответом является `iterations = 0` и никакнх дополнительных вычислений не требуется.

Обратите внимание, что в программе не производится полное вычисление аргумента: вычисляется значение `x*x + y*y`, а затем проверяется, меньше ли оно 4. Такой подход позволяет упростить вычисления, поскольку заранее известно, что 2 — это корень квадратный из 4, и, следовательно, нет необходимости вычислять квадратные корни в самой программе:

```
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;
}
```

Максимальное число повторений данного цикла равно 40.

Получив значение для данной точки в переменной `iterations`, используем оператор `switch` для выбора символа, который будет выводиться в этой точке. Вместо сорока возможных значений здесь участвуют всего четыре разных символа, которые определяются с помощью оператора взятия по модулю (`%`) таким образом, что значения 0, 4, 8... выводятся одним символом, значения 1, 5, 9... другим символом и т. д.:

```
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;
}
```

Обратите внимание, что вместо оператора `Console.WriteLine()` используется `Console.Write()`, поскольку после вывода очередного символа не требуется переходить на новую строку.

Необходимость перейти на новую строку возникает после выполнения самого вложенного цикла, и здесь просто выводится символ конца строки, для чего применяется уже знакомая escape-последовательность:

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

В итоге каждая строка программы отделена от предыдущей и выровнена надлежащим образом.

Конечный результат данного приложения оказывается не очень красивым, но весьма впечатляющим; во всяком случае, он демонстрирует, насколько полезным может быть использование циклов и ветвления в программах.

Прекращение выполнения цикла

Иногда возникают ситуации, когда требуется более тонкий контроль за выполнением кода в цикле. В C# для этих целей предусмотрено четыре команды, с тремя из которых вы уже познакомились:

- `break` — приводит к немедленному завершению цикла.
- `continue` — приводит к немедленному завершению текущего прохода цикла (вычисления продолжаются со следующего прохода).
- `goto` — позволяет выйти из цикла в помеченную точку кода (этот метод не рекомендуется использовать, если вы хотите получить легко читаемый и понимаемый код).
- `return` — выводит из цикла и из функции, в которой он находится (см. главу 6).

Команда `break` позволяет просто выйти из цикла, при этом управление передается на первую строку кода после цикла. Например:

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

Этот код выведет числа от 1 до 5, поскольку когда значение переменной `i` достигнет 6, команда `break` приведет к выходу из цикла.

Команда `continue` прекращает выполнение текущего прохода цикла, а не всего цикла. Например:

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

Третий способ предполагает использование оператора `goto`, с которым вы познакомились ранее. Например:

```
int i = 1;
while (i <= 10)
{
   if (i == 6)
      goto exitPoint;
   Console.WriteLine("{0}", i++);
}
Console.WriteLine("Этот код никогда не будет выполняться!");
exitPoint:
Console.WriteLine("Этот код будет выполнен после того,
                   как произойдет выход из цикла посредством goto.");
```

Обратите внимание на то, что выход из цикла посредством команды `goto` допустим (разве что несколько неряшлив), а вот использование `goto` для входа в цикл извне запрещено.

Бесконечные циклы

Возможно — в результате ошибки или преднамеренно — создать циклы, которые никогда не завершатся: так называемые **бесконечные** циклы. В качестве самого простого примера можно привести следующий цикл:

```
while (true)
{
   // код, выполняющийся в цикле
}
```

Иногда такой цикл бывает полезен, тем более что из него всегда можно выйти, например, с помощью оператора `break`.

Однако в тех случаях, когда создание бесконечного цикла происходит по недосмотру, последствия могут оказаться очень неприятными. Рассмотрим следующий цикл, который очень напоминает цикл `for` из предыдущего раздела:

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

Здесь увеличение переменной цикла `i` происходит в последней строке цикла, т. е. после оператора `continue`. Поэтому если управление передается на оператор `continue` (а это произойдет, когда переменная `i` станет равной 2), следующая итерация будет исполняться с тем же значением переменной `i`, опять пройдет проверка текущего значения `i`, начнет выполняться итерация и т. д. Это приведет к зависанию приложения. Обратите внимание, что в этом случае существует возможность выйти из зависшего приложения обычным способом, поэтому не придется перезагружать компьютер.

Итоги

В этой главе мы расширили наши познания в области программирования, рассмотрев различные конструкции, которые можно использовать в программах. Правильное использование таких конструкций окажется очень важным, когда вы начнете создавать более сложные приложения; а они будут встречаться сплошь и рядом на протяжении всей этой книги.

Мы уделили некоторое время изучению булевой логики, а также достаточно подробно рассмотрели логику побитовых операций. Эти вопросы имеют очень большое значение, когда дело доходит до использования ветвления и организации циклов в программах. Представляется необходимым хорошо изучить операторы и приемы, представленные в настоящей главе.

Ветвление позволяет выполнять код по условию, что в сочетании с использованием циклов дает возможность создавать достаточно сложные конструкции. Когда цикл расположен внутри другого цикла, который в свою очередь находится внутри оператора `if`, помещенного внутри третьего цикла, то начинаешь понимать, почему так важно пользоваться отступами при написании программ. Если сдвинуть весь код к левому краю экрана, тут же возникнут трудности с его восприятием и еще большие трудности появятся при его отладке. Очень важно научиться использовать отступы уже на данном этапе: позднее вы сумеете это оценить! Конечно, VS выполняет эту работу за нас, однако в любом случае представляется полезным самостоятельно соблюдать отступы при наборе текста кода.

Упражнения

1. Если имеются две целые переменные `var1` и `var2`, то какую проверку необходимо выполнить, чтобы убедиться, что одно из них (но не оба) больше 10?

2. Напишите приложение, использующее логику из упражнения 1, которое получает два числа от пользователя и выводит их на экран, но если оба числа больше 10, отбрасывает их и запрашивает новые.

3. Что неправильно в приведенном ниже коде?

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

4. Измените приложение для вычисления множества Мандельброта таким образом, чтобы оно запрашивало у пользователя границы рисунка и выводило выбранную часть рисунка. Написанная нами программа выводит такое количество символов, которое умещается на одной строке консольного приложения; вам следует предусмотреть приведение любого выбранного участка изображения к этим же размерам с тем, чтобы максимизировать видимую область.



Дополнительные сведения о переменных

Теперь, когда вы больше узнали о языке C#, самое время вернуться назад и рассмотреть наиболее сложные темы, касающиеся переменных.

Первой темой, к обсуждению которой мы приступим в этой главе, является **преобразование типов**, которое описывает преобразование значений одного типа в значения другого типа. Мы уже несколько раз сталкивались с примерами таких преобразований, однако теперь рассмотрим эту тему с формальной точки зрения. Усвоение этой информации поможет намного лучше понять, что происходит при использовании в одном выражении различных типов переменных (преднамеренно или случайно), и более точно управлять манипуляциями над данными. Это позволит не отвлекаться при написании кода и избежать неприятных сюрпризов.

Вы познакомитесь с новыми типами переменных:

- **Перечислимый тип** — тип переменных, которые принимают дискретные значения из множества, определяемого пользователем, и которые могут нести дополнительную смысловую нагрузку.
- **Структуры** — сложные типы переменных, которые создаются из определяемого пользователем набора других типов переменных.
- **Массивы** — это тип, который содержит большое количество переменных одного и того же типа и который позволяет осуществлять индексный доступ к отдельным значениям.

Эти типы оказываются несколько сложнее, чем те, с которыми вы знакомились до сих пор, однако они могут существенно упростить нам жизнь.

После того как мы завершим изучение этих тем, мы рассмотрим еще один полезный аспект — манипуляции над строками.

Преобразование типов

В начале книге мы уже обсуждали тот факт, что любые данные, независимо от своего типа, представляют собой последовательность двоичных битов, т. е. последовательность, состоящую из нулей и единиц. Собственно значение переменной зависит от того, каким образом эти биты интерпретируются. Простейшим примером такого подхода может служить тип `char`, который содержит номер символа из множества символов Unicode. Этот номер хранится абсолютно так же, как и число типа `ushort`: в обоих случаях это число, лежащее в диапазоне от 0 до 65 535.

Однако в большинстве случаев оказывается, что для различных типов переменных используются разные способы представления данных. Отсюда следует, что даже в тех ситуациях, когда имеется физическая возможность перенести последовательность битов из одной переменной в переменную какого-либо другого типа (например, обеим переменным для хранения требуется одинаковая область памяти, или у той переменной, в которую осуществляется запись, оказывается достаточно места для размещения всех исходных битов), результаты могут совершенно не совпадать с теми, которые вы ожидали!

Вместо такой переписи битов из одной переменной в другую приходится использовать **преобразования типов** данных.

Существует два вида преобразования типов:

- **Неявное преобразование** — когда преобразование типа A в тип B возможно при любых обстоятельствах, а правила проведения такого преобразования достаточно просты, чтобы доверить его компилятору.
- **Явное преобразование** — когда преобразование типа A в тип B возможно только при определенных обстоятельствах либо правила преобразования настолько сложны, что возникает необходимость выполнения каких-либо дополнительных действий.

Мы последовательно рассмотрим оба типа.

Неявные преобразования

Неявное преобразование не требует со стороны программиста никакой дополнительной работы и никакого дополнительного кода. Рассмотрим пример кода:

```
var1 = var2;
```

Такое присваивание может включать в себя проведение неявного преобразования типов в том случае, если тип переменной `var2` допускает проведение неявного преобразования в тип переменной `var1`; однако с тем же успехом в этом присваивании могут быть задействованы две переменные одного и того же типа, и тогда никакого неявного преобразования типов вообще не требуется.

Давайте рассмотрим пример.

Значения типа `ushort` и `char` являются фактически взаимозаменяемыми, поскольку оба они предполагают хранение чисел в диапазоне между 0 и 65 535. Можно осуществлять преобразования этих типов друг в друга неявно, что иллюстрируется следующим кодом:

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

В этом коде значение, хранящееся в переменной `sourceVar`, переносится в переменную `destinationVar`. Затем две команды `Console.WriteLine()` дают следующий выходной поток:

```
sourceVar val: a
destinationVar val: 97
```

Хотя в обеих переменных хранится одна и та же информация, она интерпретируется совершенно по-разному в зависимости от типа переменной.

Тип	Допускает безопасное преобразование в
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

Для простых типов существует большое количество неявных преобразований; для типов bool и string неявных преобразований не существует, зато они имеются у численных типов. В таблице слева, приведенной для справки, показаны преобразования числовых типов, которые могут выполняться компилятором неявно (не забывайте, что значения типа char хранятся в виде чисел, поэтому мы рассматриваем char как числовый тип).

Не волнуйтесь: эту таблицу не придется заучивать наизусть, поскольку существует довольно простой способ определять, какие именно преобразования компилятор может выполнять неявно. Если вернуться к главе 3, то там можно найти диапазоны допустимых значений для каждого простого числового типа. Правило неявных преобразований для рассматриваемых типов можно сформулировать следующим образом: любой тип A, диапазон допустимых значений которого целиком находится внутри диапазона допустимых значений типа B, может быть преобразован неявным образом в этот тип.

Это правило имеет очень простое обоснование. Если вы попытаетесь присвоить переменной некоторое значение, а это значение окажется выходящим за пределы тех значений, которые она может принимать, то возникнут определенные проблемы. Так, например, в переменной типа short могут храниться значения до 32 767, а максимально допустимое значение типа byte ограничено 255, поэтому если мы попытаемся преобразовать значение типа short в значение типа byte, это может создать проблему. Если значение типа short лежит в диапазоне между 256 и 32 767, то оно просто не уместится в одном байте.

Однако в том случае, если вам точно *известно*, что значение типа short меньше 255, должна ведь существовать какая-то возможность выполнить такое преобразование, не правда ли?

Простой ответ на этот вопрос гласит: да, такая возможность существует. Немного более развернутый ответ таков: да, такая возможность существует, но для этого необходимо воспользоваться **явным** преобразованием. При выполнении явного преобразования вы как бы говорите: "Хорошо, я предупрежден об опасности подобных действий, и я принимаю всю ответственность за возможные последствия на себя".

Явные преобразования

Как и предполагает их название, явные преобразования выполняются в тех случаях, когда мы явным образом обращаемся к компилятору с просьбой преобразовать значение одного типа в другой. Именно поэтому для их выполнения требуется наличие дополнительного кода, причем формат этого кода может варьироваться в зависимости от конкретного метода выполнения преобразования. Прежде чем приступить к рассмотрению кода, применяемого для выполнения явных преобразований, давайте узнаем, что произойдет в том случае, если для явного преобразования *не будет* использоваться никакого кода.

В качестве примера, несколько модифицировав код из предыдущего раздела, попытаемся преобразовать значение типа `short` в значение типа `byte`:

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

При попытке выполнить компиляцию этого кода будет выведено следующее сообщение об ошибке:

```
Cannot implicitly convert type 'short' to 'byte'
(Невозможно неявно преобразовать тип 'short' в 'byte')
```

К счастью для нас, компилятор C# умеет обнаруживать отсутствие необходимых явных преобразований!

Для того чтобы такой код можно было откомпилировать, нужно добавить код, который выполнял бы это преобразование явно. Наиболее простой способ добиться этого в данном контексте — это привести переменную типа `short` к переменной типа `byte`. Приведение типа, как правило, означает принудительный перевод данных из одного типа в другой и имеет следующий синтаксис:

```
(destinationType) sourceVar
```

Эта запись приведет к преобразованию значения переменной *sourceVar* к типу *destinationType*.

> *Обратите внимание, что такое возможно только в определенных ситуациях. Для типов, мало похожих друг на друга или вообще не имеющих ничего общего, преобразование типов с помощью приведения, вероятнее всего, не будет определено.*

Теперь можно модифицировать наш пример таким образом, чтобы добиться преобразования из типа `short` в тип `byte`:

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

Результатом будет:

```
sourceVar val: 7
destinationVar val: 7
```

А что происходит в тех случаях, когда мы пытаемся загнать в переменную значение, которое там не помещается? Давайте модифицируем нашу программу:

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

В этом случае мы получим следующий результат:

```
sourceVar val: 281
destinationVar val: 25
```

Итак, что же произошло? Посмотрим на двоичное представление этих двух чисел, а также максимально допустимого значения для типа `byte`, равного 255:

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

Легко заметить, что потеряли самый левый бит исходного значения. Немедленно возникает вопрос: каким образом это можно предсказать? Совершенно очевидно, что будут встречаться ситуации, когда придется явно приводить один тип к другому, и было бы неплохо знать заранее, какие данные при этом будут утрачиваться. Не умея этого делать, можно столкнуться с серьезными ошибками, например, в приложениях для работы с банковскими счетами.

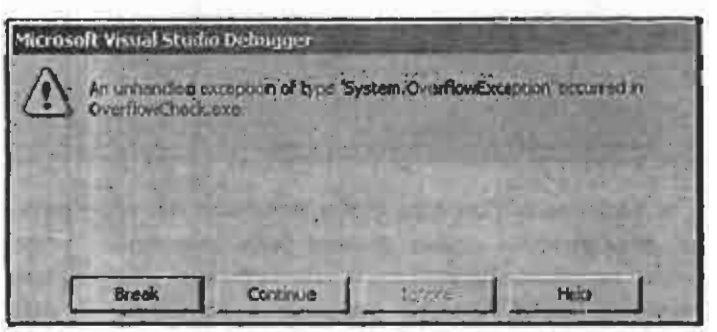
Один из способов предвидеть потерю битов — проверка исходного значения и его сравнение с известными предельными значениями новой переменной. Каким образом это можно выполнить, будет рассказано в следующей главе при объяснении логики работы приложений и управления порядком выполнения. Но имеется и другой способ, который заставляет систему во время работы уделять особое внимание данному преобразованию. Попытка присвоить переменной слишком большое значение приводит к **переполнению**, и это как раз та ситуация, которую требуется контролировать.

Контекст проверки на переполнение задается с помощью двух ключевых слов: `checked` (проверяемое) и `unchecked` (непроверяемое), которые используются следующим образом:

```
checked(выражение)
unchecked(выражение)
```

Итак, давайте потребуем осуществить проверку на переполнение в предыдущем примере:

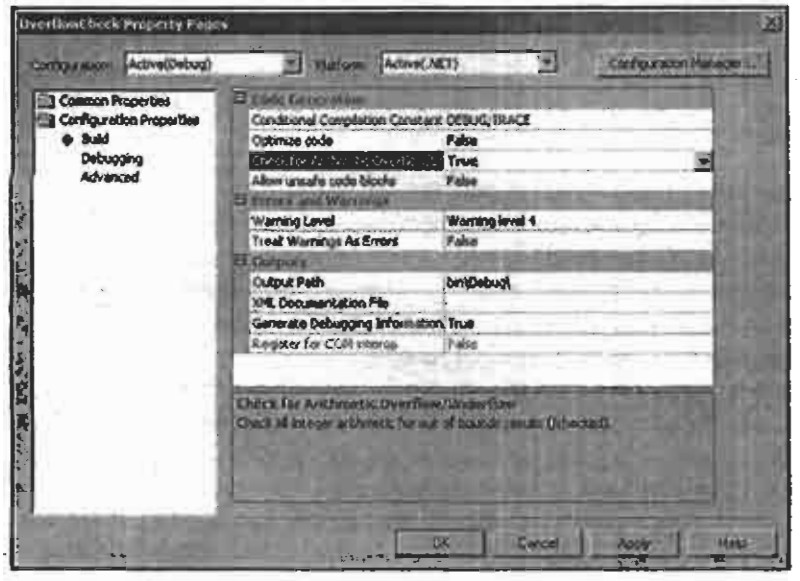
```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```



Процесс выполнения кода будет прерван появлением сообщения об ошибке (см. рис. слева) (этот пример откомпилирован в проекте с названием `OverflowCheck`).

Однако в том случае, если заменить ключевое слово `checked` на `unchecked`, получится тот же результат и никакой ошибки не возникает. Такой вариант идентичен поведению по умолчанию, с которым мы познакомились выше.

Наряду с этими ключевыми словами имеется возможность поменять конфигурацию приложения, чтобы оно работало так, как если бы в любом выражении данного типа использовалось ключевое слово `checked`, кроме тех выражений, в которых явно использовано ключевое слово `unchecked` (другими словами, мы можем менять установки, которые используются по умолчанию для выполнения проверок на переполнение). Для этого потребуется изменить свойства нашего приложения в VS. Щелкните правой кнопкой мыши по названию проекта в окне Solution Explorer

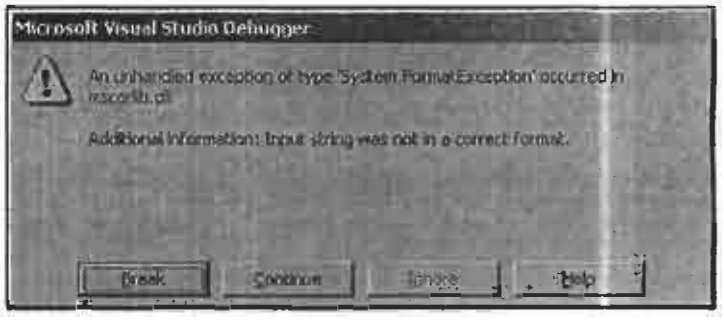


и выберите опцию Properties. Откройте папку Configuration Properties в левой части окна; появится список из трех пунктов: Build, Debugging и Advanced. Свойство, которое нам требуется изменить, является одной из установок компилятора, поэтому необходимо выбрать пункт Build. В расположенную вверху группу свойств (Code Generation) входит опция под названием Check for Arithmetic Overflow/Underflow (см. рис. слева). По умолчанию эта установка имеет значение `false`, однако при замене ее на `true` приложение будет работать в режиме `checked`, подробно описанном выше.

Выполнение явных преобразований с помощью команд преобразования

Способ явного преобразования, использовавшийся во многих предыдущих примерах, несколько отличается от того, с чем мы столкнулись в настоящей главе. Прежде мы осуществляли преобразования строковых значений в числа посредством таких команд, как `Convert.ToDouble()`, которые совершенно очевидно не смогут работать с любой строкой.

Если, например, попытаться преобразовать строку "Number" в число типа `double` с помощью `Convert.ToDouble()`, то в процессе выполнения кода будет открыто диалоговое окно с сообщением: "Необрабатываемая исключительная ситуация типа 'System.FormatException' в mscorlib.dll. Дополнительная информация: Входная строка имеет неверный формат." (см. рис. справа).



Как видите, операция не выполнена. Для того чтобы подобный тип преобразования мог осуществиться, передаваемая строка должна быть допустимым представлением числа и это число не должно приводить к переполнению. Допустимым называется такое представление числа, которое содержит необязательный знак (плюс или минус), ноль или несколько цифр, необязательную десятичную точку, за которой следует одна или несколько цифр, а также необязательный символ "e" или "E", за которым следуют необязательный знак и одна или несколько цифр, и *ничего более, кроме пробелов* (располагающихся перед или после этой последовательности). Используя все эти дополнительные опции, мы можем распознавать строки типа -1.245e-24 как число.

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

Существует много явных преобразований, которые могут быть определены таким же образом (см. табл. слева).

Здесь *val* может быть переменной почти любого типа (а если она не может быть обработана данными командами, то компилятор выдаст соответствующее сообщение).

К сожалению, как можно заметить из приведенной таблицы, имена команд преобразования несколько отличаются от названий типов, принятых в C#; например, для того чтобы произвести преобразование в тип `int`, следует использовать команду `Convert.ToInt32()`. Это объясняется тем, что имена в командах взяты из пространства имен `System` .NET Framework и не являются родными для C#. Зато благодаря этому их можно использовать из любых других языков, совместимых с .NET.

Необходимо отметить один важный момент: в процессе таких преобразований *всегда* осуществляется контроль за переполнением, при этом ключевые слова `checked` и `unchecked`, а также установки, заданные для всего проекта, влияния не имеют.

Рассмотрим пример, который охватывает многие типы преобразований, описанные в этом разделе.

Практикум: использование преобразований типов

1. Создайте новое консольное приложение с именем `Ch05Ex01` в директории `C:\BegCSharp\Chapter5`.

2. Добавьте следующий код в `Class1.cs`:

```
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");

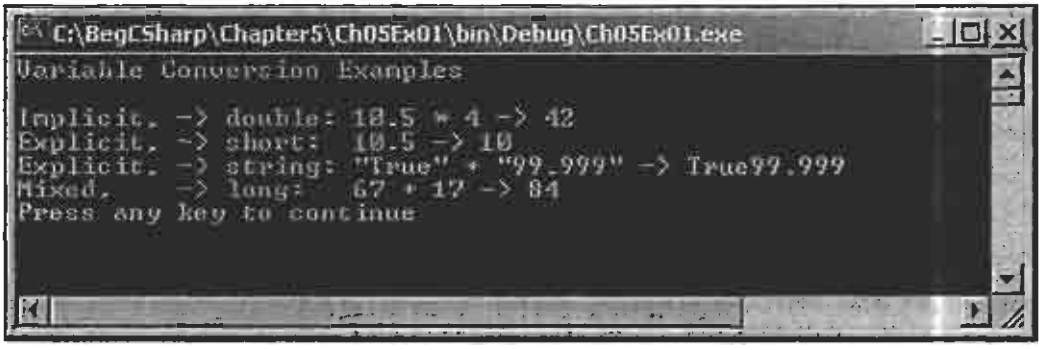
    doubleResult = floatVal * shortVal;
    Console.WriteLine("Implicit, -> double: {0} * {1} -> {2}", floatVal,
                      shortVal, doubleResult);

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

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

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

3. Запустите программу:



Как это работает

В этом примере содержатся все типы преобразований, с которыми вы успели познакомиться. Они выполняются как при обычном присваивании (это отражено в приведенных выше коротеньких примерах), так и в выражениях. Нам нужно рассмотреть оба случая, поскольку выполнение *любого* не унарного оператора — а не только оператора присваивания — может привести к преобразованию типов. Например:

```
shortVal * floatVal
```

Здесь перемножаются значения типа `short` и типа `float`. В подобных ситуациях, когда выполнение преобразования явно не указано, если это возможно, будет выполняться неявное преобразование. В данном примере единственным преобразованием, имеющим смысл, является преобразование `short` в значение типа `float` (поскольку преобразование из `float` в `short` должно задаваться явно), поэтому именно оно и будет использовано.

Однако при желании мы можем переопределить такое поведение:

```
shortVal * (short)floatVal
```

> *Это вовсе не означает, что при выполнении данной операции будет получено значение типа `short`. Поскольку результатом перемножения двух значений типа `short` с большой вероятностью может оказаться число, превосходящее 32 767 (максимально допустимое значение, которое может содержаться в типе `short`), то в действительности эта операция возвращает значение типа `int`.*

Явные преобразования, определяемые с помощью этого синтаксиса изменения типов, имеют такое же старшинство, как и другие унарные операторы (например, оператор ++, используемый в качестве префикса): их старшинство максимально.

Когда встречается выражение, в котором используются смешанные типы, то преобразования осуществляются по мере выполнения отдельных операторов в соответствии с их старшинством. Это означает, что могут происходить некоторые "промежуточные" преобразования. Например:

```
doubleResult = floatVal + (shortVal * floatVal);
```

Первый оператор, который должен выполняться в данном случае,— это оператор *, который, как только что было сказано, приведет к преобразованию значения `shortVal` к типу `float`. Затем будет выполняться оператор +, которому не потребуется производить никаких преобразований, поскольку он будет работать с двумя значениями типа `float` (`floatVal` и значением типа `float`, получившимся в результате выполнения операции `shortVal * floatVal`). Наконец, значение типа `float` будет преобразовано в значение типа `double` при выполнении оператора =.

На первый взгляд весь этот процесс преобразования типов может показаться сложным, однако при разбивке выражения на составные части с учетом порядка выполнения операций все становится совершенно понятно.

Сложные типы переменных

Итак, к настоящему моменту мы рассмотрели все простые типы переменных, которые используются в C#. Кроме них, в C# существует три более сложных (но очень полезных) типа переменных:

- Перечислимый тип
- Структуры
- Массивы

Перечислимый тип

Все типы, которые рассматривались до сих пор (за исключением типа `string`), имеют четко определенное множество допустимых значений. Это множество может быть настолько большим (как, например, у типа `double`), что его можно рассматривать в качестве континуума, однако все равно это фиксированное множество. В качестве простейшего примера можно привести тип `bool`, который может принимать только одно из двух значений: `true` или `false`.

Существует большое количество ситуаций, когда требуется переменная, принимающая значение из фиксированного множества. Например, может возникнуть необходимость в использовании переменной типа `orientation` (ориентирование), которая принимает одно из значений: `north` (север), `south` (юг), `east` (восток) или `west` (запад).

В подобных ситуациях очень полезным может оказаться **перечислимый** тип. Он позволяет сделать как раз то, что требуется для переменной `orientation`: определить тип, который принимает одно значение из конечного множества задаваемых нами значений.

Все, что для этого требуется сделать,— это создать свой собственный перечислимый тип, который будет называться `orientation` и для которого будут существовать четыре возможных значения, перечисленных выше. Обратите внимание, что это является дополнительным шагом: мы не просто объявляем переменную некоторого типа, мы сначала объявляем и подробно описываем создаваемый пользователем тип и только после этого получаем возможность объявить переменную данного типа.

Определение перечислимых типов

Перечислимый тип описывается с помощью ключевого слова enum следующим образом:

```
enum имяТипа
{
   значение1,
   значение2,
   значение3,
   . . .
   значениеN
}
```

Затем объявляются переменные этого типа:

```
имяТипа имяПеременной;
```

Им присваиваются конкретные значения:

```
имяПеременной = имяТипа.значение;
```

Перечислимый тип обладает **базовым типом** (underlying type), который используется для хранения. Любое из значений, которые этот тип может принимать, будет храниться в памяти как значение базового типа (по умолчанию это тип int). Однако существует возможность задать в качестве базового другой тип, добавив к описанию типа имя его базового типа:

```
enum имяТипа : базовыйТип
{
   значение1,
   значение2,
   значение3,
   . . .
   значениеN
}
```

Перечислимые типы могут использовать в качестве базовых следующие типы: byte, sbyte, short, ushort, int, uint, long и ulong.

По умолчанию каждому значению перечислимого типа автоматически присваивается соответствующее значение базового типа, начиная с нуля, в том порядке, в котором они описаны. Другими словами, *значение1* получит базовое значение 0, *значение2* — 1, *значение3* — 2 и т. д. Для того чтобы переопределить такой порядок, следует использовать оператор = и фактические базовые значения для каждого перечислимого значения:

```
enum имяТипа : базовыйТип
{
   значение1 = фактическоеЗначение1,
   значение2 = фактическоеЗначение2,
   значение3 = фактическоеЗначение3,
   . . .
   значениеN = фактическоеЗначениеN
}
```

Кроме того, существует возможность задавать идентичные базовые значения для нескольких перечислимых значений, используя одно значение как базовое значение другого:

```
enum имяТипа : базовыйТип
{
   значение1 = фактическоеЗначение1,
   значение2 = значение1,
```

```
    значение3,
    . . .
    значениеN = фактическоеЗначениеN
}
```

Всем значениям, оставшимся неуказанными, будут автоматически присвоены базовые значения; для этого используется последовательность, начинающаяся со значения, на единицу большего последнего явно заданного значения. В вышеприведенном объявлении *значение3* получит базовое значение, равное *значение1* + 1.

Обратите внимание, что это может привести к возникновению непредвиденных проблем, когда какие-либо из базовых значений, заданных после определения вроде *значение2 = значение1*, окажутся идентичными другим базовым значениям. Так, например, в следующей программе базовое значение для *значение4* и для *значение2* будет одним и тем же:

```
enum имяТипа : базовыйТип
{
    значение1 = фактическоеЗначение1,
    значение2,
    значение3 = значение1,
    значение4,
    . . .
    значениеN = фактическоеЗначениеN
}
```

Это совершенно нормально, если, конечно, вы хотели добиться именно такого эффекта.

Обратите также внимание, что присваивание значений циклическим образом приведет к возникновению ошибки:

```
enum имяТипа : базовыйТип
{
    значение1 = значение2,
    значение2 = значение1
}
```

Теперь давайте посмотрим, каким образом все это можно использовать.

Практикум: использование перечислимых типов

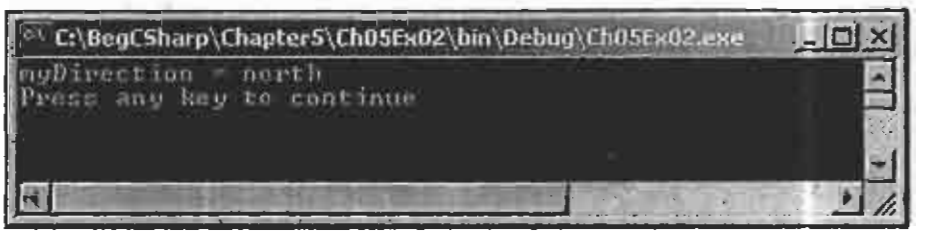
1. Создайте новое консольное приложение с именем `Ch05Ex02` в директории `C:\BegCSharp\Chapter5`.

2. Добавьте следующий код в `Class1.cs`:

```
namespace Ch05Ex02
{
    enum orientation : byte
    {
        north = 1,
        south = 2,
        east  = 3,
        west  = 4
    }
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
```

```
        {
        orientation myDirection = orientation.north;
            Console.WriteLine("myDirection = {0}", myDirection);
        }
    }
}
```

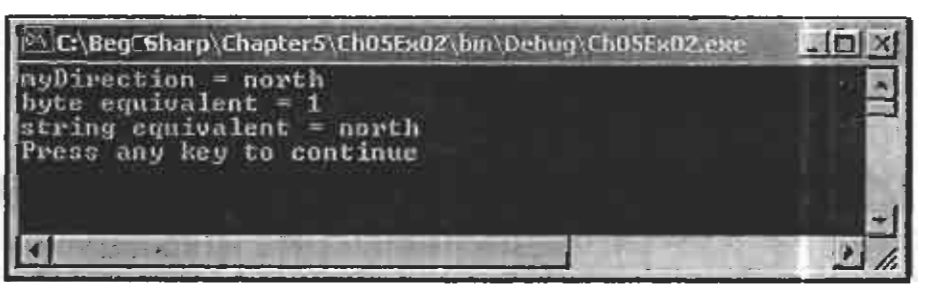
3. Запустите приложение. На консоли появится следующий выходной поток:



4. Выйдите из приложения и измените программу 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);
Console.WriteLine("string equivalent = {0}", directionString);
```

5. Повторно запустите приложение:



Как это работает

В данной программе определяется и используется перечислимый тип с именем `orientation`. Первое, на что следует обратить внимание: код с описанием типа помещен в пространство имен `Ch05Ex02`, а не в то место, где находится остальной код. Это сделано потому, что в процессе выполнения программы не происходит последовательного выполнения строк определения, как это имеет место с другими строками кода приложения. Выполнение приложения начинается с определяемого нами места, причем приложение имеет доступ к новому типу, поскольку принадлежит тому же пространству имен.

Первый вариант нашего примера демонстрирует основной метод создания переменной нового типа, присваивание ей значения и вывод этого значения на экран.

Затем мы модифицировали код, чтобы показать преобразование перечислимых значений в другие типы. Заметьте, что в таких случаях необходимо использовать

явное преобразование. Несмотря на то что базовым типом типа orientation является тип byte, нам все равно приходится использовать явное приведение типа (byte) для преобразования значения переменной myDirection в тип byte:

```
directionByte = (byte)myDirection;
```

Такое же явное приведение типа требуется и для обратного преобразования, когда тип byte нужно преобразовать в тип orientation. Например, для преобразования переменной типа byte с именем myByte в тип orientation и присвоения получившегося значения переменной myDirection можно воспользоваться следующим кодом:

```
myDirection = (orientation)myByte;
```

Естественно, в данном случае необходимо действовать очень аккуратно, поскольку не каждое из допустимых значений переменной типа byte соответствует какому-либо из описанных значений переменной типа orientation. В переменных типа orientation вполне могут храниться и другие значения типа byte, и, хотя это не приведет к непосредственному возникновению ошибки, логика дальнейшего выполнения приложения может быть нарушена.

Для получения строкового значения из значения перечислимого типа можно использовать Convert.ToString():

```
directionString = Convert.ToString(myDirection);
```

Приведение (string) в данном случае не будет выполняться, поскольку здесь требуется более сложная обработка, нежели простой перенос данных, хранящихся в переменной перечислимого типа, в переменную типа string.

В качестве альтернативы можно воспользоваться командой ToString() самой переменной. Следующий код даст точно такой же результат, что и использование Convert.ToString():

```
directionString = myDirection.ToString();
```

Обратное преобразование значения типа string в значение перечислимого типа также возможно, однако для этого потребуется немного более сложный синтаксис. Для преобразований такого рода существует специальная команда Enum.Parse(), которая используется следующим образом:

```
(перечислимыйТип)Enum.Parse(typeof(перечислимыйТип),
                                значениеСтрокиПеречислимогоТипа);
```

Здесь задействован еще один оператор — typeof, который позволяет получать тип своего операнда. Мы могли бы воспользоваться им для нашего перечислимого типа orientation следующим образом:

```
string myString = "north";
orientation myDirection = (orientation)Enum.Parse(typeof(orientation,
    myString);
```

Естественно, далеко не каждое строковое значение будет соответствовать какому-либо значению типа orientation. Использование значения, которому не соответствует ни одно из значений перечислимого типа, приведет к возникновению ошибки. Как и все в C#, эти значения чувствительны к регистру, поэтому мы получим ошибку даже в том случае, если наша строка будет соответствовать перечислимому значению во всем, кроме регистра (например, если переменной myString будет присвоено значение "North" вместо "north").

Структуры

Еще одним типом переменной, который мы будем рассматривать, является `struct` (сокращение от слова structure — "структура"). **Структуры** вполне соответствуют своему названию: это структуры данных, которые составлены из информации различного характера, возможно, из данных различных типов. Они позволяют программистам описывать свои собственные типы переменных, для которых данная структура является базовой. В качестве примера предположим, что нам необходимо хранить информацию о маршруте из одной точки в другую, включающую направление и расстояние в милях. Для простоты в качестве направления будет использоваться только одна из четырех сторон света (так что для описания направления вполне подойдет перечислимый тип `orientation`), а длина маршрута будет представлена типом `double`.

В этом случае мы могли бы использовать две разные переменные, написав уже знакомый нам код:

```
orientation myDirection;
double      myDistance;
```

В таком подходе с использованием двух переменных нет ничего плохого, однако намного удобней (особенно если маршрутов много) хранить всю эту информацию в одном месте.

Описание структур

Для описания структур используется ключевое слово `struct`:

```
struct <имяТипа>
{
   <объявлениеЧлена>
}
```

В разделе *<объявлениеЧлена>* содержатся объявления переменных (они называются полями) практически в обычном формате. Объявление каждого члена имеет следующий вид:

```
<доступ> <тип> <имя>;
```

Для того чтобы код, обращающийся к структуре, имел доступ к ее полям, в разделе *<доступ>* следует использовать ключевое слово `public` (общий). Например:

```
struct route
{
   public orientation direction;
   public double      distance;
}
```

После того как мы описали структуру, мы можем использовать ее для описания переменных соответствующего типа:

```
route myRoute;
```

Для получения доступа к отдельным полям этой составной переменной следует использовать точку:

```
myRoute.direction = orientation.north;
myRoute.distance = 2.5;
```

Давайте рассмотрим использование этого типа на примере.

Практикум: использование структуры

1. Создайте новое консольное приложение с именем Ch05Ex03 в директории C:\BegCSharp\Chapter5.

2. Добавьте следующий код в Class1.cs:

```
namespace Ch05Ex03
{
   enum orientation : byte
   {
      north = 1,
      south = 2,
      east  = 3,
      west  = 4
   }
   struct route
   {
      public orientation direction;
      public double distance;
   }
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      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:");
            myDirection = Convert.ToInt32(Console.ReadLine());
         }
         while ((myDirection < 1) || (myDirection > 4));
         Console.WriteLine("Input a distance:");
         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);
      }
   }
}
```

3. Запустите программу, выберите направление и введите расстояние:



Как это работает

Структуры, так же как и перечислимые типы, описываются вне основного тела кода. Мы описали структуру `route` внутри объявления пространства имен, в том же месте, в котором описывается используемый данной структурой тип `orientation`:

```
enum orientation : byte
{
   north = 1,
   south = 2,
   east  = 3,
   west  = 4,
}
struct route
{
   public orientation direction;
   public double      distance;
}
```

Основное тело кода имеет схожую структуру с некоторыми примерами, встречавшимся нам ранее: запрос входной информации у пользователя и вывод ее на экран. Мы выполняем некоторую несложную проверку допустимости вводимой информации, помещая выбор направления внутрь цикла `do` и отбрасывая любую введенную информацию, не являющуюся целым числом в диапазоне от 1 до 4 (эти значения для упрощения процедуры присваивания соответствуют членам перечислимого типа).

Еще один интересный момент, на который следует обратить внимание, заключается в том, что работа с полями переменной типа `route` строится точно таким же образом, как если бы мы обращались к переменным того типа, что и эти поля. Присваивание, например, выполняется следующим образом:

```
myRoute.direction = (orientation)myDirection;
myRoute.distance = myDistance;
```

В принципе мы могли бы непосредственно присваивать введенное значение полю `myRoute.distance` без каких бы то ни было нежелательных последствий:

```
myRoute.distance = Convert.ToDouble(Console.ReadLine());
```

Наличие дополнительного шага позволяет проверять вводимые значения, хотя в самом коде такая проверка не осуществляется.

Любой доступ к полям структуры организовывается аналогичным образом. Допустимо считать, что выражения типа *переменнаяТипаСтруктуры.переменнаяТипаПоля* представляют собой переменную типа *переменнаяТипаПоля*.

Массивы

Все типы переменных, которые мы рассматривали до сих пор, обладают одним общим свойством: в них хранится единственное значение (или — в случае структур — единственный набор значений). Но в ситуациях, когда необходимо хранить большое количество однородных данных, такой подход может оказаться не совсем удобным. В таких случаях более приемлемым было бы одновременное хранение нескольких значений одного типа без использования отдельной переменной для каждого значения.

Предположим, к примеру, что вам необходимо выполнить вычисления, в которых задействованы имена всех ваших друзей. Для этого можно воспользоваться обычными переменными типа `string`, например, так:

```
string friendName1 = "Robert Barwell";
string friendName2 = "Mike Parry";
string friendName3 = "Jeremy Beacock";
```

Однако в этом случае вам потребуется затратить много усилий, особенно по мере написания кода для обработки каждой переменной в отдельности. Отсутствует и возможность пройти по всему этому списку строк в цикле.

Альтернативной возможностью является использование массивов. Массив представляет собой список индексированных переменных. Допустим, что у нас имеется массив, в котором хранятся все три имени, приведенные выше, и который называется `friendNames`. Тогда мы получаем доступ к отдельным элементам этого массива, просто указывая их индекс в квадратных скобках, как показано в следующем примере:

```
friendNames[<индекс>]
```

Индекс — это целое число, имеющее значение 0 для первого вхождения, 1 — для второго и т. д. Это означает, что у нас появляется возможность пройти все вхождения в цикле, например:

```
int i;
for (i = 0; i < 3; i++)
{
   Console.WriteLine("Имя с индексом {0}: {1}", i, friendNames[i]);
}
```

Каждый массив обладает единым **базовым типом**, т. е. все отдельные вхождения в массив имеют один и тот же тип. В данном случае массив `friendNames` имеет базовый тип `string`, поскольку он предназначается для хранения переменных типа `string`.

Вхождения массива часто называются **элементами**.

Объявление массивов

Массивы объявляются следующим образом:

```
<базовыйТип>[] <имя>;
```

В данном случае *<базовыйТип>* может быть произвольным типом, в том числе перечислимым или структурой.

Прежде чем получить доступ к массиву, его необходимо инициализировать. Но получить доступ или присвоить значение элементу массива следующим образом нельзя:

```
int[] myIntArray;
myIntArray[10] = 5;
```

Массивы могут быть инициализированы двумя способами. Мы можем либо задать полное содержимое массива в литеральной форме, либо указать размер массива и использовать для инициализации всех его элементов ключевое слово `new` (новый).

Определение значений элементов массива в литеральной форме предполагает задание списка значений, разделенных запятыми и заключенных в фигурные скобки. Например:

```
int[] myIntArray = {5, 9, 10, 2, 99};
```

В данном случае описывается массив, состоящий их пяти элементов, и всем им присваивается целое значение.

Второй способ предполагает использование следующего синтаксиса:

```
int[] myIntArray = new int[5];
```

Здесь используется ключевое слово new для явной инициализации массива и константа для определения его размера. При этом методе всем вхождениям массива будет присвоено значение по умолчанию, которое для числениых типов равно 0. Для инициализации массивов вместо констант с тем же успехом можно использовать переменные, например:

```
int[] myIntArray = new int[размерМассива];
```

При желании можно одновременно использовать оба метода:

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

В этом случае размеры массива должны совпадать с колнчеством элементов в списке. Следующая запись является недопустимой:

```
int[] myIntArray = new int[10] {5, 9, 10, 2, 99};
```

Здесь описывается массив, состоящий из 10 элементов, а определяются только 5 из них; компиляция этой записи не пройдет. Побочным эффектом такого положения вещей является то, что если мы описываем размерность массива с помощью переменной, то она должна являться константой. Например:

```
const int arraySize = 5;
int[] myIntArray = new int[arraySize] {5, 9, 10, 2, 99};
```

Если ключевое слово const будет пропущено, программа работать не будет.

Как и в случае переменных других типов, нет никакой необходимости инициализировать массив в той же строке,-в которой он объявляется. Вполне допустимой является следующая запись:

```
int[] myIntArray;
myIntArray = new int[5];
```

Теперь вы уже знаете достаточно для того, чтобы попробовать написать некоторую программу.

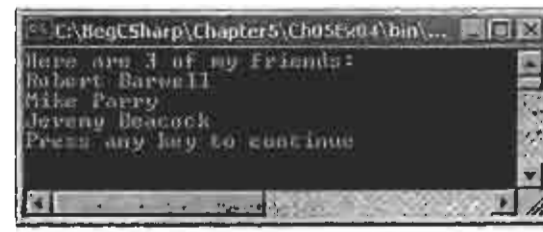
Практикум: использование массивов

1. Создайте новое консольное приложение с именем Ch05Ex04 в директории C:\BegCSharp\Chapter5.

2. Добавьте следующий код в Class1.cs:

```
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);
   for (i = 0; i < friendNames.Length; i++)
   {
      Console.WriteLine(friendNames[i]);
   }
}
```

3. Запустите программу (см. рис. справа).



Как это работает

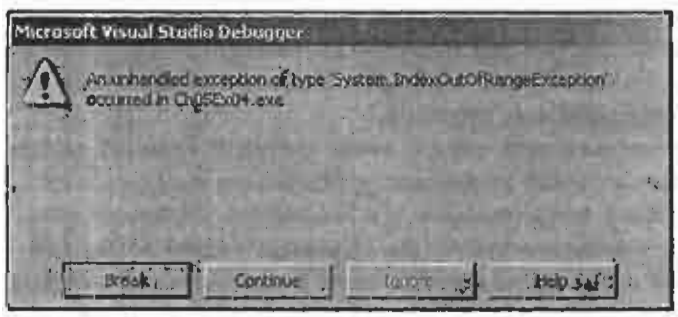
В этом коде объявляется массив типа `string`, состоящий из трех значений, которые затем выводятся на консоль с помощью цикла `for`. Обратите внимание, что существует возможность узнать общее количество элементов массива посредством параметра `friendNames.Length`:

```
Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
```

Это очень удобный способ определения размера массива.

При выводе значений массива в цикле `for` легко допустить ошибку. Например, попытаемся вместо оператора < использовать оператор <=, как показано ниже:

```
for (i = 0; i <= friendNames.Length; i++)
{
   Console.WriteLine(friendNames[i]);
}
```



Компиляция с последующим выполнением программы приведет к появлению окна с сообщением об ошибке: "Необрабатываемая исключительная ситуация типа 'выход индекса за пределы диапазона' в `Ch05Ex04.exe`" (см. рис. слева).

В приведенном случае мы попытались получить доступ к `friendNames[3]`. Как вы помните, значения индексов массива начинаются с нуля, поэтому последним элементом данного массива является `friendNames[2]`. Обращение к элементам массива, выходящим за его границу, приведет к ошибке.

На самом деле существует более удобный способ доступа ко всем элементам массива с помощью цикла `foreach`.

Циклы foreach

Цикл `foreach` позволяет обращаться ко всем элементам массива с помощью очень простого синтаксиса:

```
foreach (<базовыйТип> <имя> in <массив>)
{
   // можно использовать <имя> для доступа к очередному элементу
}
```

Этот цикл пройдет по всем элементам массива, последовательно помещая каждый из них в переменную *<имя>*; при этом опасность выхода за границы массива отсутствует. Нам не придется беспокоиться о том, сколько элементов имеется в массиве, и мы сможем воспользоваться в цикле каждым из них. Для того чтобы применить этот способ, изменим код последнего примера:

```
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)
        }
    }
```

Выходной поток этой программы будет абсолютно таким же, как и в предыдущем примере.

Основное отличие этого метода от обычного цикла `for` заключается в том, что он позволяет использовать содержимое массива в режиме "только чтение", т. е. отсутствует возможность изменять значения элементов массива. Например, написать такую программу нельзя:

```
foreach (string friendName in friendNames)
{
    friendName = "Rupert the bear";
}
```

При попытке откомпилировать этот код будет выдано сообщение об ошибке.

Многомерные массивы

Прочитав заглавие этого раздела, вы, возможно, подумаете, что мы собираемся обсуждать какое-то научно-фантастическое приложение к языку C#. На самом же деле многомерный массив — это объект, в котором для осуществления доступа к его элементам используется несколько индексов.

В качестве примера представим себе случай, когда требуется отразить высоту возвышенности по измеряемым точкам на местности. Мы можем определить такие точки двумя координатами x и y и использовать эти координаты в качестве индексов так, чтобы в массиве с названием `hillHeight` хранилась высота холма, соответствующая точке с данной парой координат. Для этого потребуется использовать многомерный массив.

Двумерный массив такого вида объявляется следующим образом:

```
<базовыйТип>[,] <имя>;
```

Описания массивов с большим числом размерностей требуют больше запятых. Например:

```
<базовыйТип>[,,,] <имя>;
```

Так объявляется четырехмерный массив.

Для присваивания значений используется аналогичный синтаксис, причем размерности отделяются одна от другой запятыми. Для того чтобы объявить и инициализировать обсуждавшийся выше двумерный массив `hillHeight` с базовым типом `double`, размерностью `x`, равной 3, и размерностью `y`, равной 4, необходима следующая строка:

```
double[,] hillHeight = new double[3,4];
```

В качестве альтернативы при начальном присваивании значений могут использоваться литеральные значения. Для этого воспользуемся вложенными блоками, взятыми в фигурные скобки и разделенными запятыми. Например:

```
double[,] hillHeight = new {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
```

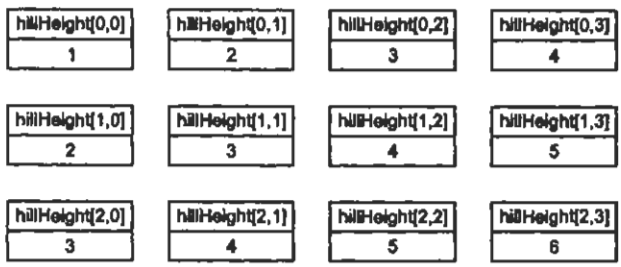
Этот массив имеет точно такие же описания размерностей, что и предыдущий, однако в данном случае значения элементов массива задаются явно.

Для того чтобы получить доступ к отдельным элементам многомерного массива, следует указать индексы этого элемента, разделенные запятыми. Например:

```
hillHeight[2,1]
```

Теперь над этим элементом можно выполнять действия так же, как и над другими элементами.

Это выражение позволяет получить доступ ко второму элементу третьего вложенного массива в соответствии с вышеприведенным описанием (в данном случае значение равняется 4). Запомните, что отсчет начинается с 0 и что первая цифра относится к вложенному массиву. Другими словами, первая цифра определяет номер пары фигурных скобок, а вторая цифра определяет соответствующий элемент внутри этой пары фигурных скобок. Визуально такой массив выглядит, как показано на рисунке слева.



Цикл `foreach` позволяет осуществлять доступ ко всем элементам многомерных массивов так же, как и в случае одномерных. Например:

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
foreach (double height in hillHeight)
{
   Console.WriteLine("{0}", height);
}
```

Элементы этого массива будут выводиться в том же порядке, в котором происходило присваивание им литеральных значений:

```
hillHeight[0,0]
hillHeight[0,1]
hillHeight[0,2]
hillHeight[0,3]
hillHeight[1,0]
hillHeight[1,1]
hillHeight[1,2]
```

и т. д.

Массивы массивов

Многомерные массивы, обсуждавшиеся в предшествующем разделе, обычно называются прямоугольными, поскольку у них каждая "строка" имеет один и тот же размер. Если использовать наш пример, то координата *y* может изменяться от 0 до 2 для любой координаты *x*.

Однако существует возможность использовать **ступенчатые** (jagged) массивы, у которых "строки" могут быть неодинакового размера. Для этого требуется массив, каждым элементом которого также является массив. При желании можно использовать массивы массивов массивов и даже более сложные конструкции. Однако это оказывается возможным только в том случае, если все массивы имеют один и тот же базовый тип.

Синтаксис, применяемый для объявления массивов массивов, подразумевает использование в объявлении массива нескольких пар квадратных скобок, например:

```
int[][] jaggedIntArray;
```

К сожалению, инициализация подобных массивов оказывается не таким простым делом, как инициализация многомерных массивов. Так, например, мы не имеем возможности использовать вслед за этим объявлением следующую запись:

```
jaggedIntArray = new int[3][4];
```

Но даже в том случае, если бы мы могли применять запись такого вида, вряд ли от этого была бы какая-нибудь польза, поскольку в случае многомерных массивов точно такого же эффекта можно достичь с меньшими затратами. Следующий код также является недопустимым:

```
jaggedIntArray = {{1, 2, 3}, {1}, {1, 2}};
```

На самом деле существуют две возможности. Можно сначала инициализировать массив, содержащий другие массивы (чтобы избежать путаницы, мы будем называть вложенные массивы субмассивами), а затем, в свою очередь, инициализировать эти субмассивы:

```
jaggedIntArray = new int[2][];
jaggedIntArray [0] = new int[3];
jaggedIntArray [1] = new int[4];
```

Можно также воспользоваться видоизмененной формой приведенного выше литерального присваивания:

```
jaggedIntArray = {new int[] {1, 2, 3}, new int[] {1}, new int[] {1, 2}};
```

Для таких неоднородных массивов можно использовать циклы `foreach`, однако чаще всего вложенные циклы применяются для того, чтобы получить доступ к реальным данным. Например, допустим, что имеется следующий неоднородный массив, который включает в себя десять других массивов, каждый из которых содержит целые значения, являющиеся делителями целых чисел из диапазона 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}};
```

Использование следующего кода:

```
foreach (int divisor in divisors1To10)
{
   Console.WriteLine(divisor);
}
```

является недопустимым, поскольку `divisors1To10` содержит в себе не элементы `int`, а элементы `int[]`. Поэтому нам придется проходить не только по самому массиву, но и непосредственно по каждому субмассиву:

```
foreach (int[] divisorsOfInt in divisors1To10)
{
   foreach (int divisor in divisorsOfInt)
   {
      Console.WriteLine(divisor);
   }
}
```

Как вы, вероятно, заметили, при использовании неоднородных массивов синтаксис очень быстро становится сложным! В большинстве случаев оказывается проще использовать либо прямоугольные массивы, либо какой-нибудь другой, более простой способ хранения данных. Однако могут возникать ситуации, в которых придется использовать именно этот метод, а дополнительные знания не повредят!

Действия над строками

До сих пор все наши операции над строками сводились к выводу строк на консоль, чтению строк с консоли и объединению строк посредством оператора +. В процессе программирования более сложных приложений вы очень скоро убедитесь, что значительная их часть предполагает различные манипуляции со строками. По этой причине нам представляется разумным потратить несколько страниц книги на наиболее распространенные способы работы со строками в C#.

Начнем с того, что переменная типа `string` — это всего лишь массив переменных типа `char`, доступных в режиме "только чтение". Иными словами, мы можем получить доступ к отдельным символам строки следующим образом:

```
string myString = "A string";
char myChar = myString[1];
```

Однако присваивать отдельные символы таким способом нельзя.

Для получения массива, доступного для записи, следует воспользоваться приведенным ниже кодом. В нем используется команда `ToCharArray()` переменной типа массива:

```
string myString = "A string";
char[] myChars = myString.ToCharArray();
```

Теперь мы можем выполнять манипуляции с массивом типа `char` обычным путем.

Строки могут использоваться и в циклах `foreach`:

```
foreach (char character in myString)
{
   Console.WriteLine("{0}", character);
}
```

Как и в случае других массивов имеется возможность узнать число элементов с помощью `myString.Length`. Таким же образом можно определить общее количество символов в строке:

```
string myString = Console.ReadLine();
Console.WriteLine("You typed {0} characters.", myString.Length);
```

Другим основополагающим способом работы со строками является использование команд в формате, аналогичном формату команды *<string>*`.ToCharArray()`. Существуют две простые, но очень полезные команды: *<string>*`.ToLower()` и *<string>*`.ToUpper()`. Они позволяют переводить всю строку целиком в нижний и верхний регистр соответственно. Чтобы оценить пользу от них, представьте ситуацию, когда вы хотите убедиться в поступлении от пользователя какого-то ответа, например, строки "yes". Если мы переведем всю введенную строку в нижний регистр, то сможем воспринять и такие строки, как "YES", "Yes", "yeS" и т. д.:

```
string userResponse = Console.ReadLine();
if (userResponse.ToLower() == "yes")
{
   // выполнение действий в случае получения ответа
}
```

Обратите внимание, что эта команда, как и другие рассматриваемые в данном разделе команды, на самом деле не изменяет ту строку, к которой применяется. Напротив, использование этой команды для некоторой строки приводит к созданию новой строки, которую можно сравнить с какой-либо другой строкой (как показано выше) или присвоить другой переменной. В роли этой переменной может выступать переменная, над которой выполняется данная операция:

```
userResponse = userResponse.ToLower();
```

Это важный момент, о котором не следует забывать, поскольку с помощью записи:

```
userResponse.ToLower();
```

многого добиться не удастся!

Давайте посмотрим, что еще мы можем сделать для облегчения интерпретации ответов пользователя. Что произойдет в том случае, если пользователь случайно включил в начало или в конец своего ответа пробел? В таком варианте вышеприведенный код не сработает. Необходимо убрать из введенной строки все пробелы, чего можно добиться посредством команды *<string>*.Trim():

```
string userResponse = Console.ReadLine();
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
{
   // выполнение действий в случае получения ответа
}
```

В этом случае мы окажемся в состоянии определить строки, подобные следующим:

```
"  YES"
"Yes "
```

Такая же команда может быть использована для удаления любых других символов, которые задаются с помощью массива типа char, например:

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = Console.ReadLine();
userResponse = userResponse.ToLower();
userResponse = userResponse.Trim(trimChars);
if (userResponse == "y")
{
   // выполнение действий в случае получения ответа
}
```

Это позволяет удалить все пробелы, символы "e" и символы "s", находящиеся в начале или в конце строки. Если предположить, что в строке отсутствуют какие-либо другие символы, то появится возможность определять строки:

```
"Yeeeees"
" y"
```

и им подобные.

Существует также возможность использовать команды *<string>*.TrimStart() и *<string>*.TrimEnd(), которые будут удалять пробелы только из начала или только из конца строки соответственно. В этих командах также имеется возможность задания массива типа char удаляемых символов.

Имеются еще две команды работы со строками, которые выполняют манипуляции с пробелами внутри строк: *<string>*.PadLeft() и *<string>*.PadRight(). Эти команды позволяют дополнять строки пробелами слева или справа до заданной длины строки. Они могут использоваться следующим образом:

```
<string>.PadX(<требуемаяДлина>);
```

Например:

```
myString = "Aligned";
myString = myString.PadLeft(10);
```

В результате к слову "Aligned", содержащемуся в переменной myString, будут добавлены три пробела слева. Этот метод может оказаться полезным для выравнивания строк, располагаемых одна над другой, в частности, при расположении строк с номерами.

Так же, как и в случае команд, предназначенных для удаления символов, описываемые команды допускают вариант использования, позволяющий задавать символ, которым будет дополняться строка. В данном случае, однако, следует задавать отдельный символ, а не массив символов. Например:

```
myString = "Aligned";
myString = myString.PadLeft(10, '-');
```

В итоге в начало строки, хранящейся в переменной myString, будет добавлено три символа тире.

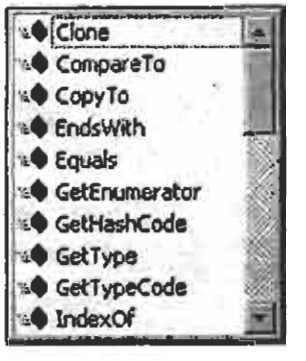
Существует большое количество других команд, выполняющих манипуляции со строками; многие из них предназначены для использования в весьма специфических ситуациях. Мы будем обсуждать их в последующих главах по мере возникновения необходимости в них. Однако перед тем как двигаться дальше, стоит рассмотреть одну из возможностей VS, на которую вы, вероятно, уже обратили внимание при чтении предыдущих глав и — особенно — данной главы.

Практикум: автоматическое завершение строк в VS

1. Создайте новое консольное приложение с именем Ch05Ex05 в директории C:\BegCSharp\Chapter5.

2. Добавьте следующий код в Class1.cs в точном соответствии с тем, как он здесь написан:

```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = {' '};
    string[] myWords;
    myWords = myString.
}
```



3. Обратите внимание, что, как только вы введете последнюю точку, немедленно раскроется окно, показанное на рисунке справа.



4. Не изменяя положение курсора, наберите символ s. Содержание окна изменится, а рядом с ним появится текст на желтом фоне (см. рис. слева).

string[] String.Split(char[] separator, int count) (+1 overloads)
Выделяет в данном экземпляре подстроки, ограниченные одним или несколькими символами, заданными в массиве, а затем переносит эти подстроки в массив String. Параметр определяет максимальное количество элементов массива, которое может быть возвращено.

5. Введите следующие символы: "(separator);".
Окно закроется, а программа примет следующий вид:

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.Split(separator);
}
```

6. Добавьте следующий код, обращая внимание на всплывающие окна:

```
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);
   }
}
```



7. Запустите программу (см. рис. слева).

Как это работает

В данном коде присутствуют два момента, на которые следует обратить внимание. Первый из них касается новой команды для работы со строками, второй относится к использованию функции автоматического завершения строк VS. Мы последовательно рассмотрим оба момента.

Команда, которую мы используем в нашей программе, — `<string>.Split()` — преобразует переменную типа `string` в массив типа `string`, разбивая ее на подстроки в указанных точках. Эти точки определяются посредством массива типа `char`, который в нашем случае содержит всего один элемент — символ пробела:

```
char[] separator = {' '};
```

Следующая часть программы получает подстроки, возникшие при разбиении исходной строки по пробелам, т. е. массив отдельных слов:

```
string[] myWords;
myWords = myString.Split(separator);
```

После этого мы проходим массив слов в цикле `foreach` и выводим каждое слово на консоль:

```
foreach (string word in myWords)
{
   Console.WriteLine("{0}", word);
}
```

Заметьте, что ни одно слово не содержит пробелов ни внутри себя, ни по краям. При использовании команды `Split()` символы, применяемые для разбиения строки (сепараторы), удаляются.

Теперь рассмотрим автоматическое завершение строк. VS — высокоинтеллектуальная система, и она вырабатывает огромное количество информации по мере того, как вы вводите свою программу. Как только вы набрали точку после слова `myString`, система определяет, что `myString` — это строка; она определяет также, что вы собираетесь задать какую-то команду, применяемую для работы со строками, и предлагает вам все возможные варианты. В этот момент при желании можно прекратить набор текста и выбрать команду, которую вы собираетесь использовать, с помощью клавиш управления курсором. По мере того как вы перемещаетесь по списку доступных команд, VS сообщает вам, что означает текущая команда, и приводит синтаксис ее использования.

Если мы продолжаем набирать текст, то VS автоматически переносит выбранную команду в начало списка возможных команд. Когда там окажется необходимая вам команда, вы можете просто продолжить ввод текста, как если бы имя этой команды было набрано вами полностью; после этого, введя символ (, вы должны задать дополнительную информацию, которую требуют некоторые команды. В этом случае VS сообщает формат, в котором должна задаваться такая информация, и представит разные варианты команд, которым требуется различное количество информации.

Данная особенность VS очень удобна, поскольку она позволяет легко находить информацию о незнакомых типах. Возможно, вам покажется интересным просмотреть все команды, существующие для типа `string`, и поэкспериментировать с ними — ничего из того, что вы будете делать, не приведет к поломке компьютера, так что играйте на здоровье!

Итоги

В этой главе определенное внимание было уделено расширению знаний о переменных и устранению некоторых имевшихся пробелов. Возможно, наиболее важной темой является преобразование типов, поскольку к ней мы будем возвращаться постоянно на протяжении всей книги. Прочно усвоив ее основные понятия, вы существенно облегчите свою жизнь в будущем!

Также было представлено несколько новых типов переменных, которые могут использоваться для более удобного хранения данных. Вы узнали, как применение перечислимых типов позволяет сделать программу более удобочитаемой за счет использования легко распознаваемых значений; как можно употреблять структуры для совместного хранения нескольких связанных элементов информации; каким образом можно группировать однородную информацию в массивы. На протяжении всей остальной книги мы не раз еще столкнемся с использованием всех этих типов.

Наконец, в этой главе описывались действия над строками и обсуждались основополагающие методы и принципы их выполнения. Для работы со строками существует большое количество команд; нами были рассмотрены только некоторые из них, однако было показано, каким образом можно вести поиск имеющихся команд в VS. Следуя этим путем и экспериментируя с различными командами, вы можете получить немало удовольствия. По крайней мере одно из приводимых ниже упражнений может быть выполнено с помощью одной или нескольких команд работы со строками, которые не обсуждались в данной главе. Попробуйте догадаться самостоятельно!

Упражнения

1. Какие из приведенных ниже преобразований не могут выполняться неявно?

   a) `int` в `short`

   b) `short` в `int`

   c) `bool` в `string`

   d) `byte` в `float`

2. Напишите программу, в которой описывается перечислимый тип `color` (цвет), использующий в качестве базового типа `short` и содержащий все цвета радуги, а также белый и черный. Может ли для такого перечислимого типа в качестве базового использоваться тип `byte`?

3. Видоизмените программу-генератор множества Мандельброта из предыдущей главы таким образом, чтобы для описания комплексных чисел использовалась следующая структура:

```
struct imagNum
{
   public double real, imag;
}
```

4. Сможет ли приведенный ниже код пройти процедуру компиляции? И если нет, то почему?

```
string[] blab = new string[5]
string[5] = 5th string.
```

5. Напишите консольное приложение, которое принимает вводимую пользователем строку и выводит строку, в которой все символы расположены в обратном порядке.

6. Напишите консольное приложение, которое принимает вводимую пользователем строку и заменяет все вхождения "no" на "yes".

7. Напишите консольное приложение, которое обрамляет двойными кавычками все слова, содержащиеся в строке.



Функции

Все программы, с которыми мы работали до сих пор, имели форму единого блока, в ряде случаев включавшем циклы, позволяющие многократно выполнять отдельные строки кода, и ветвление, позволяющее выполнять некоторые операторы по условию. Если возникала необходимость выполнить какую-либо операцию над даннымн, то нужный код должен был находиться там, где он будет работать.

Однако оказывается, что такая структура кода весьма ограниченна. Довольно часто вознпкают ситуации, когда выполнение определенных задач — напрнмер, поиск максимального элемента массива — необходимо в различных местах программы. Конечно, можно размещать идентичные (или почти идентичные) участки кода в разных частях приложения каждый раз, когда в этом возникает необходимость, однако такой подход имеет свои недостатки. Изменение даже одной небольшой деталн, касающейся выполнения общей задачи (например, исправление некоторой ошибки), потребует внесения изменений во многих местах, возможно, разбросаниых по всему приложению, а размер таких приложений бывает очень большим. Пропуск одиого места может привести к непоправимым последствням и невериому использоваиию всего приложения.

Решением такого рода проблем является прнменение **функций**. Функции в C# — это средство, позволяющее выполнять некоторые участки кода в произвольном месте приложения.

> *Функции особого типа, которые мы будем рассматривать в настоящей главе, известны под названием **методов**. Однако при программировании в среде .NET этот термин имеет особое значение, которое станет понятным по мере изучения этой книги, поэтому на данный момент мы будем избегать его использования.*

Например, мы можем написать функцию, которая осуществляет поиск максимального элемента масснва. В результате появится возможность использовать эту функцию из произвольиой точки программы, причем в каждом случае будут выполняться одни и те же строки кода. Поскольку мы должны иаписать этот код только один раз, то изменеиия, которые потребуется в него внести, повлияют на все вычисления, где бы этот код ни использовался. Такую функцию можно представить себе как **повторно используемый** код.

Функции также обладают тем преимуществом, что они позволяют делать программу более удобочитаемой, и мы получаем возможность группировать вместе логически связанные между собой части программ. Поступая таким образом, можно сделать тело самого приложения иебольшим, поскольку решение внутренних

задач приложения будет осуществляться отдельно. Это напоминает способ, посредством которого в VS можно соединять различные участки кода, используя режим схематического просмотра программ (outline view), что позволяет придать приложению более логичную структуру.

Функции могут также использоваться для создания **многоцелевых** программ, которые выполняют одни и те же операции над различными данными. Мы имеем возможность передавать функциям информацию, с которой они должны работать, в виде параметров и получать результаты работы функции в виде **возвращаемых значений**. В приведенном выше примере можно передать функции в качестве параметра массив, в котором осуществляется поиск, и получить элемент массива с максимальным значением в качестве возвращаемого значения. Отсюда следует, что мы можем каждый раз использовать одну и ту же функцию для работы с различными массивами. Параметры и возвращаемое значение функции вместе называются **сигнатурой** функции.

В настоящей главе мы начнем с того, что рассмотрим способы описания и использования простых функций, которые не получают и не возвращают никаких значений. После этого мы перейдем к способам передачи и получения информации от функций.

Затем мы обратимся к **области действия переменных**. Каким образом данные приложения на C# локализуются в различных участках кода — этот вопрос оказывается особенно важным, когда программа разбивается на множество отдельных функций.

После этого мы перейдем к подробному рассмотрению функции, играющей значительную роль в приложениях на C#,— к функции `Main()`. Вы узнаете, каким образом можно применять встроенные в нее алгоритмы для использования **аргументов командной строки**, которая позволяет передавать информацию приложениям в процессе их запуска.

Далее вы познакомитесь с некоторыми дополнительными возможностями структур, о которых шла речь в предыдущей главе, а именно с использованием функций в качестве полей структур.

В заключение главы мы сосредоточимся на двух более сложных темах: на **перегрузке функций** и на **делегатах**. Перегрузка функций — это способ, позволяющий иметь несколько функций с одинаковым именем, но с отличающимися сигнатурами. Делегат — тип переменной, который допускает непрямое использование функции. Один и тот же делегат может быть использован для вызова любой функции, имеющей указанную сигнатуру, что позволяет осуществлять выбор из нескольких различных функций в процессе выполнения.

Описание и использование функций

В данном разделе рассказывается, каким образом можно включать функции в состав приложений, а затем использовать (**вызывать**) их из кода. Сначала будут рассмотрены простые функции, не обменивающиеся данными с вызывающим их кодом, а затем мы перейдем к более сложному использованию функций.

Итак, давайте сразу начнем с примера.

Практикум: описание и использование простой функции

1. Создайте новое консольное приложение с именем `Ch06Ex01` в директории `C:\BegCSharp\Chapter6`.

2. Добавьте следующий код в Class1.cs:

```
class Class1
{
   static void Write()
   {
      Console.WriteLine("Text output from function.");
   }

   static void Main(string[] args)
   {
      Write();
   }
}
```

3. Запустите программу:



Как это работает

Следующие четыре строки кода описывают простую функцию с именем Write():

```
static void Write()
{
   Console.WriteLine("Text output from function.");
}
```

Код, содержащийся в данной функции, просто выводит некоторый текст в консольном окне. Однако это не столь важно, поскольку в настоящий момент нас в большей степени интересуют механизмы, лежащие в основе описания и использования функции.

В данном случае описание функции состоит из:

- Двух ключевых слов: static и void
- Имени функции, за которым располагаются параметры: Write()
- Участка выполняемого кода, заключенного в фигурные скобки

Код, который используется для описания функции Write(), выглядит почти так же, как и код самого приложения:

```
static void Main(string[] args)
{
   . . .
{
```

Это объясняется тем, что весь код, который мы создавали до сих пор (не считая описания типов), представляет собой часть некоторой функции. Эта функция — Main() — выполняет (как упоминалось в главе 3) функцию точки входа в консольное приложение. Когда запускается приложение, написанное на C#, то происходит вызов содержащейся в нем функции точки входа, а когда эта функция заканчивает свою работу, выполнение приложения прерывается. Любой выполняемый код, написанный на C#, должен иметь **точку входа**.

Единственное отличие между функцией `Main()` и функцией `Write()` (не считая тех строк кода, которые в них содержатся) заключается в том, что в круглых скобках, расположенных за именем функции `Main`, находится некоторый код. Этот код служит для задания параметров; к более подробному его обсуждению мы вернемся несколько позже.

Как уже упоминалось ранее, обе функции — и `Main()`, и `Write()` — описываются с использованием ключевых слов `static` (статический) и `void` (отсутствует). Ключевое слово `static` имеет отношение к понятиям объектно-ориентированного программирования, к его рассмотренню мы перейдем ниже. На данном этапе от вас требуется запомнить только то, что все функции, которые будут задействованы в приложениях данного раздела, *обязательно должны* использовать это ключевое слово.

Ключевое слово `void`, напротив, объяснить очень просто. Оно указывает, что функция не возвращает никакого значения. Далее в этой главе будет рассказано, что необходимо писать в тех случаях, когда у функции имеется возвращаемое значение.

Продолжая рассматривать наше приложение, мы обнаруживаем код, который осуществляет вызов функции:

```
Write();
```

Он состоит нз нменн функции, за которым помещаются пустые круглые скобки. Когда выполненне программы достигнет этой точки, начнет выполняться код, содержащийся в функции `Write()`.

> *Обратите внимание, что использование круглых скобок как при описании функции, так и при ее вызове является обязательным. Можете попробовать их убрать — код не пройдет компиляцию.*

Возвращаемые значения

Самый простой способ обмена данными с фуикцнями — использование возвращаемого значения. Функции, в которых применяются возвращаемые значения, точно так же обладают *численным* значеннем, как и любые переменные, используемые прн вычислении выражений. Аналогично переменным возвращаемые значения обладают типом.

Например, можно опнсать функцию с именем `getString()`, возвращаемое значение которой будет иметь тип `string`, н использовать ее в своей программе:

```
string = myString;
myString = getString();
```

С другой стороны, можно опнсать функцию с именем `getVal()`, которая будет возвращать значение типа `double`, и использовать ее в математическом выражении:

```
double myVal;
double multiplier = 5.3;
myVal = getVal() * multiplier;
```

Еслн функция должна обладать возвращаемым значеннем, то необходимо внестн два нзменения:

- В описании функцин вместо ключевого слова `void` указать тип возвращаемого значения.
- По завершении всех вычислений в функции использовать ключевое слово `return` и передать возвращаемое значенне вызывающему коду.

Синтаксис кода для рассматриваемого типа функций консольного приложения будет выглядеть следующим образом:

```
static <возвращаемыйТип> <имяФункции>()
{
   . . .
   return <возвращаемоеЗначение>;
}
```

Единственным ограничением в данном случае является требование, гласящее, что *<возвращаемоеЗначение>* должно иметь тип *<возвращаемыйТип>* или же должна существовать возможность его неявного преобразования в этот тип. Вообще говоря, *<возвращаемыйТип>* может быть любым, включая самые сложные типы из числа рассмотренных ранее.

В простейшем случае это может выглядеть следующим образом:

```
static double getVal()
{
   return 3.2;
}
```

Однако в реальной жизни возвращаемые значения обычно являются продуктом выполняемых функцией некоторых вычислений, поскольку того же результата можно достигнуть простым использованием переменной типа `const`.

Когда при выполнении программы достигается оператор `return`, управление немедленно передается обратно в вызывающий код. Никакие строки кода после этого оператора выполняться не будут. Отсюда, однако, совершенно не следует, что в теле функции оператор `return` обязательно должен быть последним. Он может быть использован и раньше, например, при ветвлении по какому-либо условию. Включение оператора `return` в цикл `for`, в блок `if` или в какую-нибудь другую структуру приведет к немедленному окончанию выполнения как этой структуры, так и всей функции в целом. Например:

```
static double getVal()
{
   double checkVal;
   // присваивание переменной checkVal некоторого значения,
   // полученного в результате некоторых вычислений.
   if (checkVal < 5)
      return 4.7;
   return 3.2;
}
```

В данном случае будет возвращено одно из двух значений — в зависимости от значения переменной `checkVal`.

Имеется единственное ограничение: оператор `return` должен выполняться до того, как будет достигнута закрывающая фигурная скобка } данной функции. Следующий код не является допустимым:

```
static double getVal()
{
   double checkVal;
   // присваивание переменной checkVal значения,
   // полученного в результате некоторых вычислений.
   if (checkVal < 5)
      return 4.7;
}
```

Если `checkVal >= 5`, то не встретится ни одного оператора `return`, а это запрещено. Все ветви должны оканчиваться этим оператором.

И последнее замечание: оператор `return` может применяться в функциях, объявленных с использованием ключевого слова `void` (у них отсутствует какое-либо возвращаемое значение). В таких случаях функция просто прекращает работу. Поэтому при использовании оператора `return` будет ошибкой размещать возвращаемое значение между ключевым словом `return` и следующей за ним точкой с запятой.

Параметры

Если функция должна получать параметры, то необходимо задать:

- Список принимаемых функцией параметров в ее описании, а также типы этих параметров
- Совпадающий список параметров при каждом вызове функции

Это предполагает использование следующего кода:

```
static <возвращаемыйТип> <имяФункции>(<типПараметра> <имяПараметра>, ...)
{
   . . .
   return <возвращаемоеЗначение>;
}
```

Здесь может быть произвольное число параметров, для каждого из которых указываются тип и имя. В качестве разделителя между параметрами ставятся запятые. Каждый из параметров доступен внутри данной функции в качестве переменной.

Например, следующая простая функция принимает два параметра типа `double` и возвращает их произведение:

```
static double product(double param1, double param2)
{
   return param1 * param2;
}
```

Теперь перейдем к рассмотрению более сложного примера.

Практикум: обмен данными с функцией

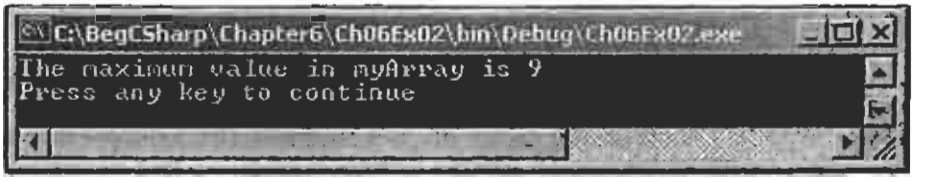
1. Создайте новое консольное приложение с именем `Ch06Ex02` в директории `C:\BegCSharp\Chapter6`.

2. Добавьте следующий код в `Class1.cs`:

```
class Class1
{
   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);
        }
    }
```

3. Запустите программу:



Как это работает

Данный код содержнт функцию, которая решает обсуждавшуюся в начале этой главы задачу. Функция принимает в качестве параметра массив целых чисел и возвращает нанбольшее из них. Ее описание имеет следующий вид:

```
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;
}
```

Данная функция — `MaxValue()` — имеет один параметр, который описан как массив типа `int` с именем `intArray`. Возвращаемое значение также имеет тип `int`. Определение максимального зиачення представляет собой несложную задачу. Локальной целой переменной с именем `maxVal` в качестве начального значения присваивается первый элемент масснва, а затем производится сравнение этого значения последовательно со всеми остальными элементами. Еслн текущий элемент больше, чем значение переменной `maxVal`, то текущее значение `maxVal` заменяется на это значение. Когда выполнение цикла завершено, переменная `maxVal` содержит наибольшее значенне данного массива, которое и возвращается оператором `return`.

Код, расположенный в `Main()`, объявляет и инициалнзирует простой целый массив, который будет использоваться совместно с функцией `MaxValue()`:

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
```

При вызове функции `MaxValue()` значение присваивается переменной `MaxVal` типа `int`:

```
int maxVal = MaxValue(myArray);
```

Затем мы выводим это значение на экран с помощью `Console.WriteLine()`:

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```

Соответствие параметров

При вызове функции ее параметры должны в точности соответствовать ее описанию. Необходимо совпадение типов параметров, их количества и порядка их следования. Это означает, что, например, функция:

```
static void myFunction(string myString, double myDouble)
{
   . . .
}
```

не может быть вызвана с использованием строки:

```
myFunction(2.6, "Hello");
```

В данном случае мы пытаемся передать значение типа `double` в качестве первого параметра, а значение типа `string` — в качестве второго, что не соответствует порядку, в котором эти параметры содержатся в описании функции. Нельзя использовать и такую строку:

```
myFunction("Hello");
```

поскольку в ней передается только один параметр типа `string` вместо двух обязательных параметров.

Попытка воспользоваться любой из этих двух строк для вызова функции приведет к ошибке при компиляции, так как компилятор налагает требование точного соответствия сигнатурам вызываемых функций.

Возвращаясь к предыдущему примеру, такое требование означает, что функция `MaxValue()` может использоваться только для получения максимального целого из массива целых чисел. Если мы изменим код в `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);
}
```

то такой код не пройдет процедуру компиляции из-за неверного типа параметра.

Ниже, в разделе, посвященном перегрузке операторов, вы познакомитесь с очень полезным способом, позволяющим решить эту проблему.

Массивы параметров

В C# предусмотрена возможность задания одного (и только одного) специального параметра функции. Этот параметр, который обязательно должен располагаться последним, известен под названием **массива параметров**. Он позволяет при обращении к функциям использовать переменное количество параметров. Массив параметров описывается посредством ключевого слова `params`.

Массивы параметров служат одним из способов упрощения кода, поскольку благодаря им не приходится передавать массивы из вызывающего кода. Напротив, можно передать несколько параметров одного и того же типа, которые помещаются в массив для последующего использования внутри функции.

При описании функции с массивом параметров применяется следующий код:

```
static <возвращаемыйТип> <имяФункции>(<п1Тип> <п1Имя>, ... ,
                                       params <тип>[] <имя>)
{
   . . .
```

```
    return <возвращаемоеЗначение>;
}
```

Для того чтобы вызвать эту функцию, потребуется следующий код:

```
<имяФункции>(<п1>, ..., <значение1>, <значение2>, ...);
```

В данном случае *<значение1>*, *<значение2>* и т. д.— это значения типа *<тип>*. Они используются для инициализации массива с именем *<имя>*. Никаких ограничений на количество параметров, которые могут быть здесь заданы, не существует; единственное предъявляемое к ним требование — они все должны быть одного типа *<тип>*. Можно вообще не задавать ни одного параметра.

Эта последняя особенность делает массивы параметров полезными, в частности, при заданнн некоторой дополнительной информацин, которая будет использоваться функцией. Например, допустим, что имеется функция `getWord()`, которая принимает значение типа `string` в качестве первого параметра и возвращает первое слово из этой строки:

```
string firstWord = getWord("This is a sentence.");
```

Переменной `firstWord` будет присвоена строка `"This"`.

Мы можем добавить в функцию `getWord()` параметр params, который позволит выбирать и возвращать другое слово по его индексу:

```
string firstWord = getWord("This is a sentence.", 2);
```

Если исходить из предположения, что нумерация слов начинается с 1, то в результате такого вызова переменной `firstWord` будет присвоена строка `"is"`.

Можно ограничить количество возвращаемых символов, введя третий параметр, что совершенно допустимо для параметра `params`:

```
string firstWord = getWord("This is a sentence.", 4, 3);
```

В этом случае переменной `firstWord` будет присвоена строка `"sen"`.

Давайте разберем пример полностью.

Практикум: обмен данными с функцией (часть 2)

1. Создайте новое консольное прнложение с нменем `Ch06Ex03` в директории `C:\BegCSharp\Chapter6`.

2. Добавьте следующий код в `Class1.cs`:

```
class Class1
{
   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);
   }
}
```



3. Запустите программу (см. рис. слева).

Как это работает

В этом примере функция `sumVals()` описана с использованием ключевого слова `params`, что позволяет ей принимать произвольное количество параметров типа `int` (и никакого другого):

```
static int sumVals(params int[] vals)
{
   . . .
}
```

Код в этой функции проходит в цикле все значения массива `vals` и суммирует их, возвращая получившийся результат.

Мы вызываем эту функцию из `Main()` с пятью целыми параметрами:

```
int sum = sumVals (1, 5, 2, 9, 8);
```

Однако можно с тем же успехом вызвать эту функцию, передавая ей ноль, один, два или сто параметров целого типа: никаких ограничений на количество задаваемых параметров не существует.

Передача параметров по ссылке и по значению

Во всех предыдущих функциях применялись параметры, передаваемые по **значению**. Другими словами, мы передавали значение в переменную, которая затем использовалась внутри функции. Любые изменения, которые эта переменная претерпевала внутри функции, *не оказывали никакого влияния* на параметр, использованный при вызове функции. В качестве примера рассмотрим функцию, которая удваивает передаваемый ей параметр и выводит результат на экран:

```
static void showDouble(int val)
{
   val *= 2;
   Console.WriteLine("val doubled = {0}", val);
}
```

В этой функции происходит удвоение передаваемого параметра. Если же мы вызовем эту функцию следующим образом:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

то выходной поток, выведенный на консоль, будет таким:

```
myNumber = 5
val doubled = 10
myNumber = 5
```

Вызов функции `showDouble()` с переменной `myNumber` в качестве параметра не оказывает никакого влияния на переменную `myNumber`, описанную в `Main()`, несмотря на то, что параметр, которому присваивается ее значение, удваивается.

Все это хорошо, но если мы *пожелаем*, чтобы значение переменной `myNumber` было изменено, возникнет проблема. Конечно, можно воспользоваться функцией, которая возвращает новое значение переменной `myNumber`, и вызвать ее следующим образом:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Однако в таком коде довольно трудно разобраться; кроме того, такой способ не позволяет отразить изменения значений переменных, используемых в качестве параметров (поскольку функции всегда возвращают только одно значение).

Вместо этого следует передавать параметр по **ссылке**. Это означает, что функция будет работать именно с той переменной, которая использовалась при вызове функции, а не с переменной, которая имеет то же значение. Следовательно, любые изменения переменной, использовавшейся функцией, отразятся на значении переменной, использовавшейся в качестве параметра. Для этого при описании параметра необходимо воспользоваться ключевым словом `ref`:

```
static void showDouble(ref int val)
{
   val *= 2;
   Console.WriteLine("val doubled = {0}", val);
}
```

Вторично ключевое слово `ref` необходимо использовать при вызове функции (это обязательное требование, поскольку тот факт, что параметр описан как `ref`, является неотъемлемой частью сигнатуры функции):

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

В этом случае на консоль будет выведен следующий текст:

```
myNumber = 5
val doubled = 10
myNumber = 10
```

На этот раз переменная `myNumber` была изменена в функции `showDouble()`.

Для переменных, используемых в качестве параметров типа `ref`, имеются два ограничения. Во-первых, поскольку существует вероятность, что в результате вызова функции значение вызываемого по ссылке параметра будет изменено, то при вызове функции *запрещается использовать переменные типа* `const`. Отсюда следует, что следующий вызов является недопустимым:

```
const int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Во-вторых, необходимо использовать заранее инициализированную переменную. C# не гарантирует, что параметр типа `ref` будет инициализирован той функцией, в которой он используется. Следующий код также является недопустимым:

```
int myNumber;
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Выходные параметры

В дополнение к возможности передавать параметры по ссылке мы можем указать, что данный параметр является выходным: в описание такого параметра включается ключевое слово `out`, используемое так же, как и ключевое слово `ref` (в качестве модификатора параметра в описании функции и при вызове функции). Фактически этот способ предоставляет почти такие же возможности, что и передача параметров по ссылке, в том смысле, что значение параметра после выполнения функции попадает в переменную, использовавшуюся при вызове этой функции. Имеются, однако, и существенные отличия. Хотя использовать переменную, которой не присвоено начальное значение, в качестве параметра типа `ref` недопустимо, она может применяться в качестве параметра типа `out`. Более того, параметр типа `out` будет рассматриваться как не имеющий начального значения самой функцией, в которой он используется. Это означает, что хотя передача переменной, которой присвоено некоторое значение, в качестве параметра типа `out` является допустимой, однако в процессе выполнения функции хранящееся в этой переменной значение будет утрачено.

В качестве примера рассмотрим расширение функции `maxValue()`, которая возвращает элемент массива с максимальным значением. Модифицируем эту функцию так, чтобы получать индекс элемента массива, содержащего наибольшее значение; в случаях, когда максимальное значение содержится в нескольких элементах, мы будем получать индекс первого максимального элемента. Для этого добавим выходной параметр:

```
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;
}
```

Эта функция может быть использована таким образом:

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxIndex;
Console.WriteLine("The maximum value in myArray is {0}",
                  maxValue(myArray, out maxIndex));
Console.WriteLine("The first occurrence of this value is at element {0}",
                  maxIndex + 1);
```

Результатом ее выполнения будет следующее:

```
The maximum value in myArray is 9
(Максимальное значение, содержащееся в массиве myArray, — 9)
The first occurrence of this value is at element 7
(Первое вхождение с таким значением найдено в элементе 7)
```

Важным моментом, на который следует обратить внимание, является необходимость использовать ключевое слово `out` при вызове функции (так же, как и ключевое слово `ref`).

Обратите внимание, что в тексте программы к значению переменной `maxIndex` при выводе ее на экран прибавляется единица. Это сделано для того, чтобы придать индексу более удобную для восприятия форму, как если бы первый элемент массива считался элементом 1, а не элементом 0.

Область действия переменных

Возможно, читая предыдущий раздел, вы задавались вопросом, зачем вообще нужно обмениваться данными с функциями. Ответ заключается в том, что в C# доступ к переменным может осуществляться только из определенных участков кода. Принято говорить, что переменная имеет определенную **область действия**, в рамках которой она является доступной.

Область действия переменной представляет собой очень важное понятие, которое лучше всего проиллюстрировать с помощью примера.

Практикум: описание и использование основной функции

1. Внесите следующие изменения в `Class1.cs` в приложении `Ch06Ex01`:

```
class Class1
{
   static void Write()
   {
      Console.WriteLine("myString = {0}", myString);
   }

   static void Main(string[] args)
   {
      string myString = "String defined in Main()";
      Write();
   }
}
```

2. Откомпилируйте код и обратите внимание на сообщение об ошибке и на предупреждение, которые появятся в списке заданий:

The name 'myString' does not exist in the class or namespace 'Ch06Ex01.Class1'
(В данном классе или пространстве имен 'Ch06Ex01.Class1' имя 'myString' отсутствует
The variable 'myString' is assigned but its value is never used
Переменой 'myString' присваивается значение, но оно нигде не используется)

Как это работает

Итак, что в данном случае неправильно? А то, что переменная `myString`, описанная в основном теле приложения (в функции `Main()`), оказывается недоступной в функции `Write()`.

Причина этой недоступности кроется в том, что у переменных имеется область, внутри которой их использование является допустимым. Эта область распространяется на тот блок кода, в котором они описаны, и на все непосредственно вложенные в него блоки кода. Блоки кода функций отделены от блоков кода, из которых они вызываются. Поэтому внутри функции `Write()` имя `myString` не считается определенным, а переменная `myString`, определенная в функции `Main()`, оказывается вне области своего действия: она может использоваться только в рамках функции `Main()`.

На самом деле в функции Write() мы могли бы использовать совершенно другую переменную myString. Попробуйте внести в программу следующие изменения:

```
class Class1
{
   static void Write()
   {
      string myString = "String defined in Write()";
      Console.WriteLine("Now in 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);
   }
}
```

Этот код успешно проходит процедуру компиляции и выдает следующий результат:



Данная программа выполняет следующие действия:

- В функции Main() описывается и инициализируется переменная с именем myString.
- Функция Main() передает управление функции Write().
- В функции Write() описывается и инициализируется переменная с именем myString, отличная от переменной с именем myString, описанной в функции Main().
- Функция Write() выводит на консоль строку, содержащую то значение переменной myString, которое ей присвоено в функции Write().
- Функция Write() возвращает управление функции Main().
- Функция Main() выводит на консоль строку, содержащую то значение переменной myString, которое ей присвоено в функции Main().

Переменные, область действия которых распространяется только на одну функцию, называются **локальными**. Существует возможность описания **глобальных** переменных, чья область действия охватывает сразу несколько функций. Внесите в программу следующие изменения:

```
class Class1
{
   static string myString;
```

```
        static void Write()
        {
            string myString = "String defined in Write()";
            Console.WriteLine("Now in Write()");
            Console.WriteLine("Local myString = {0}", myString);
            Console.WriteLine("Global myString = {0}", Class1.myString);
        }

        static void Main(string[] args)
        {
            string myString = "String defined in Main()";
            Class1.myString = "Global string";
            Write();
            Console.WriteLine("\nNow in Main()");
            Console.WriteLine("Local myString = {0}", myString);
            Console.WriteLine("Global myString = {0}", Class1.myString);
        }
    }
```

В данном случае будет получен следующий результат:



На этот раз мы включили в код еще одну переменную с именем `myString`, изменив иерархию имен в программе. Эта переменная описана следующим образом:

```
static string myString;
```

Обратите внимание, что нам вновь потребовалось использовать ключевое слово `static`. Мы не будем вдаваться в детали: на данном этапе достаточно знать, что для консольных приложений данного типа в описании глобальных переменных *обязательно использование* либо ключевого слова `static`, либо ключевого слова `const`. Если мы планируем изменять значение глобальной переменной, то обязательным является ключевое слово `static`, так как `const` запрещает любые изменения значения переменной.

Для того чтобы отличать глобальную переменную от локальных переменных с тем же именем в функциях `Main()` и `Write()`, нам необходимо классифицировать имя глобальной переменной, используя полностью квалифицированное имя, которое описано в главе 3. В данном случае мы обращаемся к глобальной переменной по имени `Class1.myString`. Обратите внимание, что это необходимо делать только в том случае, если существуют глобальная и локальная переменные с одинаковым именем; если бы локальной переменной с именем `myString` не существовало, то мы совершенно свободно могли бы использовать для обращения к глобальной переменной имя `myString` вместо `Class1.myString`. В том случае, когда используется локальная переменная, имя которой совпадает с именем глобальной переменной, говорят, что глобальная переменная является **скрытой**.

Значение глобальной переменной задается в функции Main():

```
Class1.myString = "Global string";
```

и используется в функции Write():

```
Console.WriteLine("Global myString = {0}", Class1.myString);
```

Теперь вы, вероятно, недоумеваете, почему мы всегда не прибегаем к такому способу обмена данными с фуикциями. Действительно, существуют определенные ситуации, когда этот способ передачи данных оказывается более предпочтительным, однако имеется не меньшее (если не большее) число ситуаций, когда он менее удобен. Стоит ли применять глобальные переменные, зависит от способа использования каждой конкретной функции. Проблема глобальных переменных заключается в том, что они, по большому счету, оказываются непригодными для функций "общего назначения", которые способны работать с любыми передаваемыми им данными и не ограничиваются данными, задаваемыми конкретной глобальной переменной. Несколько позже мы вернемся к подробному рассмотрению этого вопроса.

Область действия переменных и другие программные конструкции

Перед тем как пойти дальше, следует обратить внимание на то, что важность заявлений, сделанных в предшествующем разделе, далеко выходит за рамки определения области действия переменных в разных функциях. Утверждалось, что область действия переменных распространяется на блок кода, в котором они описаны, и на все блоки, непосредственно в него вложенные. Это также применимо и к другим типам блоков, например, к программным конструкциям, использующим ветвление и циклы. Рассмотрим следующую программу:

```
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);
```

В данной программе строковая переменная text является локальной для цикла for. Такой код не пройдет компиляцию, поскольку в обращении к Console.WriteLine(), которое происходит вне этого цикла, делается попытка использовать переменную text, область действия которой не распространяется за пределы цикла. Изменим 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 недопустим. Причина кроется в том, что переменные должны и описываться, и инициализироваться до того, как они будут использоваться, а переменная text инициализируется только в цикле for. Значение, присвоенное переменной

text, при выходе из цикла будет утрачено. Однако мы можем модифицировать код еще раз:

```
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 вне цикла, и мы имеем доступ к ее значению. Результат выполнения этой простой программы показан на рисунке слева.

В данном случае значение, присвоенное переменной text внутри цикла, оказывается доступным и вне его.

Как вы, вероятно, заметили, эта тема требует определенных усилий для понимания. В первый момент представляется не вполне очевидным, почему — в свете предыдущего примера — переменная text не сохраняет в качестве своего значения пустую строку, присвоенную ей перед началом цикла, для кода, расположенного после его окончания.

Причина кроется в способе выделения памяти для переменной text, да и для всех остальных переменных тоже. Простое объявление переменной некоторого простого типа не влечет за собой выполнения каких-либо существенных действий. Только тогда, когда переменным присваиваются значения, для этих значений выделяется память, в которой они будут храниться. Когда такое выделение памяти происходит внутри цикла, это значение определяется как локальное и область его действия не выходит за пределы цикла. И хотя переменная не является локализованной внутри данного цикла, к ее значению это не относится. Напротив, присваивание переменной значения вне цикла дает гарантию того, что это значение локальное для всего основного кода и что область действия переменной распространяется в том числе и на цикл. Другими словами, мы будем оставаться внутри области действия переменной до тех пор, пока не покинем блок основного кода, поэтому возможность доступа к переменной имеется и за пределами цикла.

На наше счастье, компилятор C# самостоятельно обнаруживает проблемы, связанные с областью действия переменных, и генерирует сообщения об ошибках.

В качестве заключительного замечания необходимо упомянуть о "наилучшей практике". Вообще говоря, лучше всего объявлять и инициализировать переменные перед теми блоками кода, в которых они используются. Исключением из этого правила могут быть переменные циклов, объявление которых является составной частью самого цикла. Например:

```
for (int i = 0; i < 10; i++)
{
   . . .
}
```

В данном случае, переменная i является локализованной в блоке кода, представляющем собой цикл, но это и хорошо, поскольку редко когда требуется доступ к значению счетчика цикла из внешнего кода.

Параметры и возвращаемые значения по сравнению с глобальными данными

Этот раздел посвящен более детальному рассмотрению обмена данными с функциями через механизм глобальных переменных и через механизм параметров и возвращаемых значений. Напомним, что речь идет о различиях между двумя программами. Первая из них имеет вид:

```
class Class1
{
   static void showDouble(ref int val)
   {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val);
   }

   static void Main(string[] args)
   {
      int val = 5;
      Console.WriteLine("val = {0}", val);
      showDouble(ref val);
      Console.WriteLine("val = {0}", val);
   }
}
```

Обратите внимание, что этот код несколько отличается от того кода, где в функции `Main()` нами использовалась переменная с именем `myNumber`. Это иллюстрация того факта, что локальные переменные могут обладать идентичными именами и не мешать друг другу. Это также означает, что две программы, которые приводятся здесь, оказываются очень похожими друг на друга, позволяя нам сосредоточить внимание на принципиальных отличиях и не беспокоиться об именах переменных.

Вторая программа имеет следующий вид:

```
class Class1
{
   static int val;

   static void showDouble()
   {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val);
   }

   static void Main(string[] args)
   {
      val = 5;
      Console.WriteLine("val = {0}", val);
      showDouble();
      Console.WriteLine("val = {0}", val);
   }
}
```

Результаты выполнения обеих программ совершенно идентичны.

Следует заметить, что не существует жестких правил, определяющих предпочтительность использования одного способа вместо другого: оба эти способа являются вполне допустимыми. Тем не менее существуют некоторые соображения, которые желательно учитывать при выборе одного из них.

Для начала обратите внимание на то, что (это уже отмечалось выше) версия функции `showDouble()`, которая использует глобальное значение, сможет использовать только глобальное значение переменной `val`. Применение такой версии функции принуждает нас использовать именно эту глобальную переменную, что несколько ограничивает гибкость в применении данной функции и требует постоянно копировать значение глобальной переменной в другие переменные в том случае, если необходимо сохранить результат. Кроме того, глобальные данные могут быть изменены где-нибудь в другом месте приложения, что приведет к получению непредсказуемых результатов (мы можем вспомнить о том, что значения изменены, слишком поздно).

Однако такая потеря гибкости зачастую является преимуществом. Существуют ситуации, когда применение некоторой функции планируется исключительно для одной конкретной цели и использование глобально описанных данных уменьшает вероятность того, что мы неправильно обратимся к функции, например, передав неверные параметры.

Конечно, такая простота в действительности делает программу менее понятной. Явное указание параметров позволяет увидеть, что именно подвергается изменениям, даже мельком взглянув на функцию. Если встречается вызов функции, имеющий вид `myFunction(val1, out val2)`, то сразу становится ясно, что следует обратить самое пристальное внимание на переменные `val1` и `val2` и что переменной `val2` по окончании выполнения функции будет присвоено новое значение. И наоборот, если у этой функции не имеется никаких параметров, то мы не сможем сделать предположения относительно того, над какими данными она выполняет манипуляции.

Наконец, не следует забывать о том, что использование глобальных данных не всегда оказывается возможным. Далее нам встретятся примеры программ, хранящихся в разных файлах и/или принадлежащих различным пространствам имен, которые взаимодействуют между собой посредством использования функций. В таких случаях код зачастую оказывается до такой степени разделенным, что непонятно, где именно следует хранить глобальные переменные.

Итак, оба способа обмена данными можно использовать с равным успехом. В большинстве случаев мы будем настаивать на применении параметров вместо глобальных данных, однако существуют отдельные ситуации, когда более подходящим может оказаться использование именно глобальных данных; и уж, во всяком случае, такой способ не является ошибкой.

Функция Main()

Теперь, когда вы познакомились почти со всеми простыми способами, применяемыми для создания и использования функций, давайте вернемся назад и более подробно рассмотрим функцию `Main()`.

Как было сказано ранее, функция `Main()` является точкой входа в приложение на C# и выполнение этой функции охватывает выполнение приложения. У этой функции имеется параметр — `string[] args`, однако до сих пор мы его не описывали. В данном разделе вы познакомитесь с этим параметром и узнаете, каким образом его можно применять.

Существуют четыре различные сигнатуры, которые можно использовать для функции Main():

- `static void Main()`
- `static void Main(string[] args)`

- `static int Main()`
- `static int Main(string[] args)`

При желании аргумент `args` можно опускать. Причина, по которой мы до сих пор пользовались вариантом с этим аргументом,— в том, что такая версия генерируется автоматически при создании консольного приложения в VS.

Третья и четвертая версии сигнатуры, приведенные выше, возвращают значение типа `int`, которое может быть использовано для указания на то, каким образом завершилось выполнение приложения, и которое часто используется для определения ошибки (хотя это в любом случае является обязательным). Обычно, возвращаемое значение равно 0, что означает "нормальное" окончание (т. е. приложение закончило свою работу и может быть завершено в безопасном режиме).

Параметр `args` функции `Main()` представляет собой метод, позволяющий приложению получать извне информацию, задаваемую в процессе его выполнения. Эта информация принимает форму **параметров командной строки**.

Параметры командной строки вам уже наверняка встречались. Когда производится запуск приложения из командной строки, то при этом очень часто имеется возможность непосредственно задать некоторую информацию, например, имя файла, загрузка которого требуется для выполнения приложения. В качестве примера давайте рассмотрим Windows-приложение Notepad. Запустить его можно, просто набрав **notepad** — либо в окне командного приглашения, либо в окне, которое появляется при выборе опции Run (Выполнить) из меню Start (Пуск). Однако в этих же окнах мы можем набрать нечто вроде: **notepad "myfile.txt"**. В результате Notepad либо загрузит файл `myfile.txt`, либо, если такого файла не существует, предложит его создать. В данном случае `"myfile.txt"` является аргументом командной строки. Мы можем сами написать консольное приложение, которое будет работать аналогичным образом, использовав параметр `args`.

Когда консольное приложение запускается, то все заданные параметры командной строки помещаются в массив `args`. В дальнейшем эти параметры могут использоваться по мере необходимости.

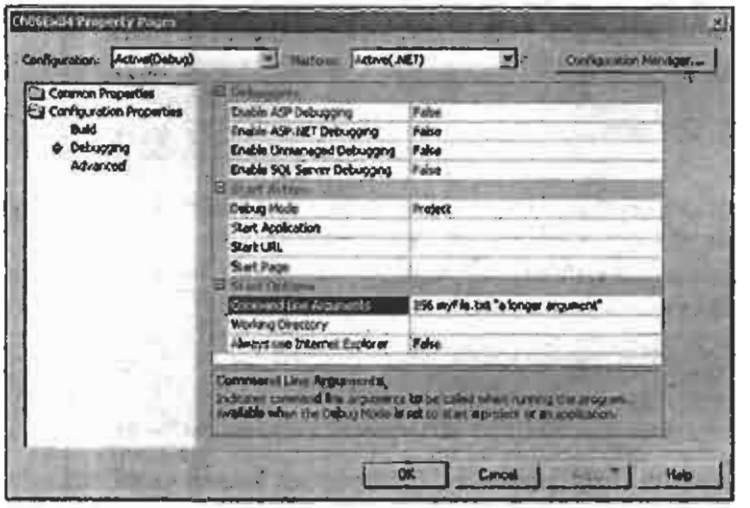
Давайте рассмотрим пример выполнения таких действий.

Практикум: аргументы командной строки

1. Создайте новое консольное приложение с именем `Ch06Ex04` в директории `C:\BegCSharp\Chapter6`.

2. Добавьте следующий код в `Class1.cs`:

```
class Class1
{
   static void Main(string[] args)
   {
      Console.WriteLine("{0} command line arguments were specified:",
                        args.Length);
      foreach (string arg in args)
         Console.WriteLine(arg);
   }
}
```



3. Откройте страницы со свойствами проекта (для этого нужно щелкнуть правой кнопкой мыши на имени проекта `Ch06Ex04` в окне Solution Explorer и в раскрывшемся меню выбрать **Properties**).

4. Выберите страницу **Configuration Properties | Debugging** и добавьте в установку **Command Line Arguments** (см. рис. слева) те аргументы командной строки, которые захотите.

5. Запустите приложение (см. рис. справа).



Как это работает

Код, использованный в данном случае, очень простой:

```
Console.WriteLine("{0} command line arguments were specified:",
                  args.Length);
foreach (string arg in args)
Console.WriteLine(arg);
```

Мы используем параметр `args` точно таким же образом, как использовали бы любой другой строковый массив. Мы не делаем с аргументами ничего особенного, просто выводим на экран все, что было задано.

В данном примере аргументы заданы посредством диалога Project Properties в VS. Это удобный способ использовать одни и те же аргументы при каждом запуске приложения из VS: в противном случае пришлось бы набирать их в командной строке при каждом запуске приложения. Такого же результата можно добиться, открыв окно командной строки в той директории, куда осуществляется вывод выходного потока приложения (`C:\BegCSharp\Chapter6\Ch06Ex04\bin\Debug`), и набрав следующий текст:

```
Ch06Ex04 256 myFile.txt "a longer argument"
```

Обратите внимание, что каждый аргумент отделяется от других с помощью пробелов; также существует возможность заключать аргументы в двойные кавычки — в том случае, если потребуется задать более длинный аргумент (двойные кавычки обязательны, если аргумент сам по себе содержит пробелы,— для того, чтобы он не рассматривался в качестве нескольких аргументов).

Функции структур

В предшествующей главе рассматривались типы структур, предназначенных для хранения в одном месте нескольких элементов информации. На самом деле возможностей у структур гораздо больше, и одной из них является способность содержать не только данные, но и функции. На первый взгляд подобное может показаться немного странным, однако на практике это приносит много пользы.

В качестве простого примера рассмотрим следующую структуру:

```
struct customerName
{
   public string firstName, lastName;
}
```

Если у нас имеются переменные типа customerName и нам необходимо вывести на консоль полное имя, мы будем вынуждены конструировать это имя из его составных частей. Например, для переменной типа customerName с именем customer можно использовать следующий синтаксис:

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine("(0) (1)", customer.firstName, customer.lastName);
```

Имея возможность включать в структуры функции, мы можем упростить эту процедуру за счет централизованного выполнения часто встречающихся задач. В данном случае это можно сделать следующим образом:

```
struct customerName
{
   public string firstName, lastName;

   public string Name ()
   {
      return firstName + " " + lastName;
   }
}
```

Эта функция очень похожа на другие функции, рассматриваемые в настоящей главе, за исключением того, что в ней отсутствует модификатор static. Причины этого отсутствия станут понятны позже, а пока достаточно просто запомнить, что это ключевое слово для функций структур не требуется. Мы можем воспользоваться этой функцией следующим образом:

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine(customer.Name());
```

Этот синтаксис гораздо проще и понятнее, чем предшествующий.

Необходимо отметить, что функция Name() имеет непосредственный доступ к полям структуры firstName и lastName. В рамках структуры customerName они могут считаться глобальными.

Перегрузка функций

В начале этой главы вы узнали, как сравнивать сигнатуру функции при ее вызове. При этом необходимо иметь отдельные функции для работы с переменными различных типов. Наличие оператора перегрузки подтверждает, что последнее действительно является проблемой. Этот оператор предоставляет возможность создавать одновременно несколько функций, имеющих одинаковое имя, но работающих с параметрами различных типов.

Например, ранее мы использовали следующую программу, в которой содержалась функция с именем MaxValue():

```
class Class1
{
   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);
   }
}
```

Эта функция может использоваться только для массивов значений типа `int`. Мы могли бы создать функции с другими именами, предназначенные для работы с параметрами других типов, переименовав вышеприведенную функцию как-нибудь вроде `IntArrayMaxValue()` и добавив функции наподобие `DoubleArrayMaxValue()` для работы с другими типами. В качестве альтернативы мы можем просто включить в нашу программу 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;
}
...
```

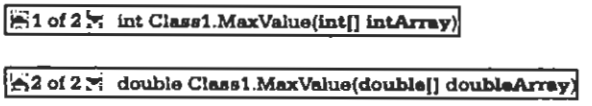
Разница между двумя функциями заключается в том, что эта функция работает со значениями типа `double`. Имя функции — `MaxValue()` — оказывается тем же самым, однако сигнатура (это принципиально) отличается. Было бы ошибкой описать две функции с одинаковым именем и одинаковой сигнатурой, однако поскольку в данном случае сигнатуры различны, то все нормально.

Теперь у нас имеются две версии функции `MaxValue()`, которые принимают массивы типа `int` и массивы типа `double` и возвращают максимальное значение типа `int` или типа `double` соответственно.

Красота такой формы программы в том, что не требуется явно указывать, которую из этих двух функций мы собираемся использовать. Мы просто задаем массив-параметр, и это приводит к выполнению того варианта, который соответствует типу используемого параметра.

На данном этапе стоит отметить еще одну высокоинтеллектуальную черту VS. Если в нашем приложении имеются две одноименные функции, описанные выше, и мы наберем это имя, например, в `Main()`, то VS выведет доступные варианты перегрузки данной функции. Если будет набрано следующее:

```
double result = MaxValue(
```



то VS выведет информацию по обеим версиям функции `MaxValue()`, которые мы можем просмотреть с помощью кнопок "стрелка вверх" и "стрелка вниз" (см. рис. слева).

При перегрузке функций учитываются все аспекты, касающиеся их сигнатур. Существует возможность, например, описать две различные функцин, одна из которых принимает параметры по значению, а другая, соответственно, по ссылке:

```
static void showDouble(ref int val)
{
   . . .
}
static void showDouble(int val)
{
   . . .
}
```

Выбор используемой версии осуществляется исключительно на основании того, имеется ли в обращении к функции ключевое слово `ref`. При следующем вызове будет использован вариант, в котором параметр передается по ссылке:

```
showDouble(ref val);
```

А такой вызов позволит передать параметр по значению:

```
showDouble(val);
```

Аналогичным образом можно описывать функции, отличающиеся числом требующихся им параметров и т. п.

Делегаты

Делегатом называется тип, который позволяет хранить ссылки на функции. Несмотря на то что звучит это несколько интригующе, механизм, лежащий в основе этой возможности, удивительно прост. Наиглавнейшее предназначение делегатов вряд ли удастся понять до тех пор, пока вы не познакомитесь с событиями и с их обработкой, однако рассмотрение самого понятия делегатов принесет огромную пользу. Когда позднее наступит время их использовать, они будут вам уже знакомы, что сделает некоторые сложные темы гораздо более легкими для понимания.

Объявление делегатов во многом напоминает объявление функций; при этом отсутствует само тело функции, но добавляется ключевое слово `delegate`. Объявление делегата определяет сигнатуру функции, состоящую из возвращаемого типа и списка параметров. После объявления делегата мы получаем возможность объявить переменную типа этого делегата. Мы можем затем инициализировать эту переменную, присвоив ей ссылку на произвольную функцию, обладающую сигнатурой, совпадающей с сигнатурой делегата. После всего этого мы получаем возможность вызывать эту самую функцию с помощью данной переменной-делегата так, как если бы последняя сама являлась этой функцией.

Теперь, когда у нас имеется переменная, ссылающаяся на функцию, мы также получаем возможность выполнять и некоторые другие операции, которые не могут быть выполнены с использованием иных средств. Например, появляется возможность передавать переменную-делегат другой функции в качестве параметра, что позволит этой функции использовать данного делегата для вызова функции, на которую он ссылается, без необходимости определять вызываемую функцию до начала выполнения программы. Давайте рассмотрим пример.

Практикум: использование делегата для вызова функции

1. Создайте новое консольное приложение с именем `Ch06Ex05` в директории `C:\BegCSharp\Chapter6`.

2. Добавьте следующий код в `Class1.cs`:

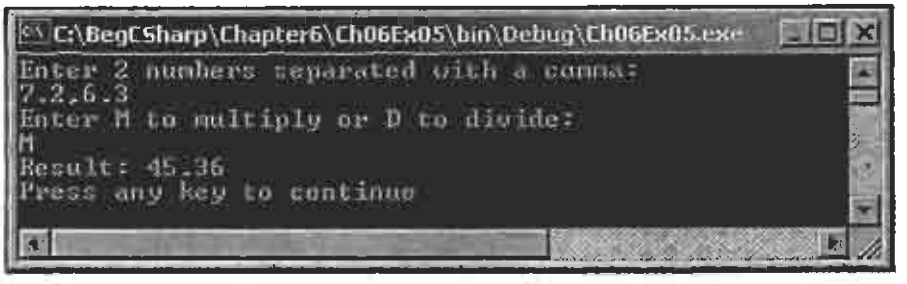
```
class Class1
{
   delegate double processDelegate(double param1, double param2);

   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:");
      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:");
      input = Console.ReadLine();
      if (input == "M")
         process = new processDelegate(Multiply);
      else
         process = new processDelegate(Divide);
      Console.WriteLine("Result: {0}", process(param1, param2));
   }
}
```

3. Запустите программу:



Как это работает

Этот код описывает делегата (`processDelegate`), сигнатура которого совпадает с сигнатурой двух функций: `Multiply()` и `Divide()`. Делегат имеет следующее определение:

```
delegate double processDelegate(double param1, double param2);
```

Ключевое слово `delegate` указывает, что данное определение является определением делегата, а не функции (это определение располагается в том же месте

программы, где могло бы находиться и определение функции). Далее приводится сигнатура, в которой задаются возвращаемое значение типа double и два параметра типа double. Используемые в сигнатуре имена являются произвольными, поэтому тип делегата и имена параметров мы можем выбирать по своему усмотрению. В данном случае мы выбрали имя делегата processDelegate, а параметры назвали param1 и param2.

Код функции Main() начинается с того, что мы объявляем переменную, воспользовавшись описанным типом делегата:

```
static void Main(string[] args)
{
   processDelegate process;
```

Затем следует абсолютно стандартный для C# код, который запрашивает два числа, разделенные запятой, и присваивает их двум переменным типа 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));
```

Обратите внимание, что для наглядности в программу не включены никакие проверки допустимости вводимой пользователем информации. В реальной программе нам пришлось бы потратить большое количество времени, проверяя допустимость тех значений, которые мы получаем для локальных переменных `param1` *и* `param2`.

Затем мы запрашиваем пользователя, следует ли умножать или делить эти числа:

```
Console.WriteLine("Enter M to multiply or D to divide:");
input = Console.ReadLine();
```

В зависимости от полученного ответа мы инициализируем переменную-делегат:

```
if (input == "M")
   process = new processDelegate(Multiply);
else
   process = new processDelegate(Divide);
```

Для присвоения переменной-делегату ссылки на функцию нам приходится использовать несколько странно выглядящий синтаксис. Так же, как и в случае присваивания значений массиву, для создания нового делегата используется ключевое слово new. После этого слова указывается тип создаваемого делегата и задается параметр, определяющий ту функцию, которую мы хотим использовать, а именно функцию Console.WriteLine(). Обратите внимание, что этот параметр не совпадает ни с параметрами типа делегата, ни с параметрами используемой функции: это уникальный синтаксис, применяемый исключительно для осуществления присваивания делегату. В качестве параметра используется просто имя той функции, которую необходимо использовать, без скобок.

Наконец, мы вызываем выбранную нами функцию с помощью делегата. Здесь применяется один и тот же синтаксис, независимо от того, на какую именно функцию ссылается делегат:

```
   Console.WriteLine("Result: {0}", process(param1, param2));
}
```

В данном случае мы обращаемся с переменной-делегатом так, как если бы она представляла собой имя функции. Однако, в отличие от функций, над этой переменной можно выполнять некоторые дополнительные операции, например, передачу ее другой функции в качестве параметра. Простой пример такого использования может выглядеть примерно следующим образом:

```
static void executeFunction(processDelegate process)
{
   process(2.2, 3.3);
}
```

Это означает, что мы получаем возможность управлять поведением функций, передавая им в качестве параметров делегатов функций, что во многом напоминает выбор используемого (snap-in). Например, может иметься функция, используемая для сортировки строкового массива по алфавиту. Существуют различные алгоритмы сортировки списков, обладающие различным быстродействием, что зависит от характеристик сортируемого списка. С помощью делегатов мы получаем возможность определять метод, который необходимо использовать, передавая функции, выполняющей сортировку, делегата той функции, в которой реализован соответствующий алгоритм сортировки.

Вариантов применения делегатов существует очень много, но, как отмечалось выше, наиболее плодотворно они используются при обработке событий. Мы будем обсуждать эту тему в главе 12.

Итоги

В данной главе был сделан достаточно полный обзор использования функций в программе на C#. Большая часть дополнительных возможностей, предоставляемых функциями (в частности, использование делегатов), являются достаточно абстрактными, и их необходимо обсуждать только с точки зрения объектно-ориентированного программирования, к рассмотрению которого мы вскоре перейдем.

Подведем итоги темам, которые обсуждались в данной главе:

- Определение и использование функций в консольных приложениях
- Обмен данными с функциями посредством параметров и возвращаемых значений
- Массивы параметров
- Передача параметров по ссылке и по значению
- Использование выходных параметров для дополнительных возвращаемых значений
- Понятие области действия переменной
- Особенности функции `Main()`, в том числе использование параметров командной строки
- Использование функций в типах структур
- Перегрузка функций
- Делегаты

Упражнения

1. В следующих двух функциях содержатся ошибки. Какие именно?

```
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 присваивания значений строковой и целой переменным соответственно, а затем выводит эти значения.

3. Создайте делегата и используйте его вместо функции `Console.ReadLine()` в запросе ввода от пользователя.

4. Модифицируйте приведенную ниже структуру, включив в нее функцию, которая возвращает суммарную стоимость заказа:

```
struct order                          // заказ //
{
   public string itemName;            // наименование //
   public int    unitCount;           // число единиц //
   public double unitCost;            // стоимость одной единицы //
}
```



Отладка и обработка ошибок

Итак, к настоящему моменту вы познакомились с основами простого программирования на C#. Перед тем как переходить к рассмотрению объектно-ориентированного программирования, которому посвящен следующий раздел книги, мы остановимся на вопросах отладки кода на C# и обработки ошибок.

Ошибки в программе — это такая вещь, от которой никогда не удастся избавиться. Независимо от того, насколько опытным является программист, у него всегда будут проскальзывать ошибки; составной частью понятия "опытный программист" как раз и является понимание этой неизбежности и готовности с ними бороться. Конечно, существуют некоторые незначительные ошибки, которые не оказывают влияния на ход выполнения приложения, такие, например, как ошибки в правописании выводимого текста и т. п. Кроме того, существуют очевидные ошибки, которые приводят к абсолютной невозможности использования приложения (их обычно называют **фатальными**). К разряду фатальных ошибок относятся как простые ошибки, которые препятствуют компиляции программы (**синтаксические** ошибки), так и более сложные ошибки, которые проявляются только в процессе выполнения кода. Однако могут встречаться еще более тонкие ошибки. Например, написанное приложение не сможет включить в базу данных запись, у которой отсутствует одно из обязательных полей, или в случае возникновения каких-либо исключительных обстоятельств включит в запись неверные данные. Ошибки, подобные этим, когда некоторый изъян содержится в логике приложения, известны под названием **семантических** ошибок (или ошибок **в логике**).

Очень часто вы будете узнавать о наличии тонких ошибок в ваших приложениях тогда, когда пользователь пожалуется, что что-то работает неправильно. В результате вам придется выполнить трассировку программы, чтобы определить, в чем именно проблема и каким образом следует изменить код, чтобы он выполнял те действия, для которых был предназначен.

В подобных ситуациях вы обнаружите, что возможности для отладки программ, предоставляемые VS, могут оказать вам ни с чем не сравнимую помощь. В первой части этой главы мы рассмотрим некоторые предлагаемые методики с точки зрения их использования для решения наиболее распространенных проблем.

Кроме этого, мы изучим способы **обработки ошибок**, имеющиеся в C#. Это позволит нам принимать соответствующие меры предосторожности в тех случаях, когда появление ошибок наиболее вероятно, и создавать код, который окажется в состоянии справляться с ошибками, могущими в противном случае стать фатальными. Такие способы представляют собой составную часть языка C#, а не отладочные возможности VS, однако и в этом случае VS предлагает некоторые весьма полезные инструменты.

Отладка в Visual Studio

Когда программы запускаются в отладочном режиме, происходит нечто большее, чем просто выполнение написанного кода. При отладке создается некоторая **символьная информация** о приложении, которая позволяет VS быть в курсе того, что именно происходит при выполнении каждой строки кода. Под символьной информацией понимается, например, информация, которая позволяет устанавливать соответствие между именами, используемыми в исходном коде, и значениями, которые существуют в скомпилированном в машинные коды приложении, не содержащем информации в удобной для чтения форме. Символьная информация помещается в файлы с расширением `.pdb`, которые вы, вероятно, встречали в директориях `Debug` на вашем компьютере. Эта информация позволяет выполнять множество полезных действий, например:

- Передачу отладочной информации системе VS
- Просмотр (и редактирование) значений рабочих переменных в процессе выполнения приложения
- Приостановку и продолжение выполнения программы
- Автоматическое прекращение выполнения программы в заранее заданных точках
- Пошаговое выполнение программы (по одной строке кода за шаг)
- Отслеживание изменений содержимого переменных в процессе выполнения приложения
- Изменение содержимого переменных в момент выполнения
- Осуществление тестовых вызовов функций

В настоящем разделе вы познакомитесь с этими методами, а также с их использованием для определения участков кода, которые работают не так, как ожидается, и внесения в них исправлений, т. е. для процесса, более известного под названием "отладка".

Все эти методы разделены на две группы по тому, каким образом они могут быть применены. Вообще говоря, процедура отладки заключается в том, чтобы либо прекращать выполнение программы, либо сохранять информацию для последующего анализа. В терминах VS приложение может находится либо в режиме выполнения, либо в **режиме останова** (break mode), означающем, что нормальное выполнение программы прервано. Сначала мы рассмотрим методы отладки программы без прекращения ее выполнения (методы периода выполнения, или нормальные методы).

Отладка без использования режима останова (в нормальном режиме)

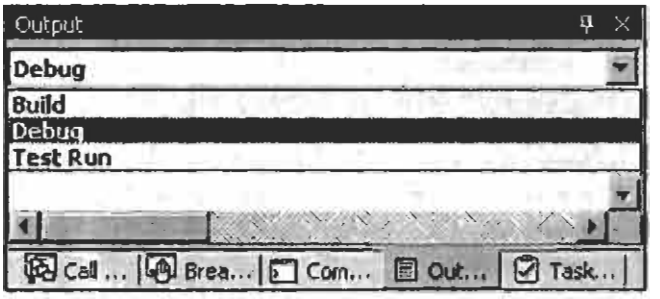
Одной из команд, которая используется нами на протяжении всей книги, является функция `Console.WriteLine()`,предназначенная для вывода текста на консоль. В процессе разработки приложений эту функцию удобно использовать для установления обратной связи с целью получения дополнительной информации о выполняемых операциях. Например:

```
Console.WriteLine("Вызываем функцию MyFunc().");
MyFunc("Выполнение некоторых действий.");
Console.WriteLine("Выполнение функции MyFunc() завершено.");
```

Этот участок кода демонстрирует, каким образом можно получать дополнительную информацию о работе функции `MyFunc()`.

Все это замечательно, однако такой прием может привести к тому, что выходной поток на консоль окажется сильно засоренным. Поэтому существует альтернатива: можно выводить текст в специально предназначенное для этого место — окно **Output** (вывод) в VS.

Как вы помните, в главе 2 вы познакомились с окном **Output**, которое по умолчанию располагается в нижней части основного окна среды разработки VS, там же, где располагается окно **Task List** (список заданий). Вы также познакомились с выводом в этом окне информации, относящейся к компиляции и выполнению кода, включая встретившиеся в процессе компиляции ошибки и т. п. Существует возможность использовать это окно для вывода спецнальной днагностической информации, которая может записываться туда напрямую. Это окно изображено на рисунке слева.



*Обратите внимание, что у этого окна имеются три режима, переход между которым осуществляется с помощью раскрывающегося списка. Мы можем переключаться между режимами **Build** (построение), **Debug** (отладка) и **Test Run** (выполнение в тестовом режиме). В режимах **Build** и **Debug** в окне показывается информация, относящаяся соответственно к периоду компиляции или к периоду выполнения. В данном разделе словосочетание "вывод в окно **Output**" будет означать "вывод в окно **Output**, находящееся в режиме **Debug**".*

В качестве альтернативы мы можем создать файл **регистрации** (logging file), в который информация будет добавляться по мере выполнения нашего приложения. Способы записи в этот файл очень напоминают способы записи текста в окно **Output**, однако они требуют понимания особенностей доступа к файловой системе из приложений на C#. На данном этапе мы не будем отвлекаться на методы доступа к файлам, поскольку нам и без того предстоит изучить большое количество материала.

Вывод отладочной информации

Вывод текста в окно **Output** в процессе выполнения является очень простой задачей. Для этого достаточно заменить обращения к функции `Console.WriteLine()` на соответствующий вызов для записи текста в нужное место. Существуют две команды, которые можно использовать с этой целью:

- `Debug.WriteLine()`
- `Trace.WriteLine()`

Обе эти команды выполняются практически одинаково, за исключением одного ключевого отличия. Первая их них работает только в том случае, если строится отладочная версия приложения, а вторая будет работать и в окончательной версии программы. В последнем случае `Debug.WriteLine()` даже не будет транслироваться,

а просто исчезнет, что, естественно, дает определенные преимущества (хотя бы то, что откомпилированная программа будет меньше по объему). На самом деле мы можем пользоваться двумя версиями одного приложения, создаваемыми из одного и того же исходного файла. Отладочная версия выводит все виды диагностической информации, в то время как основная версия не несет этой дополнительной нагрузки и не выводит эти сообщения, способные сильно раздражать пользователя!

Следует обратить внимание, что работа этих функций несколько отличается от работы функции `Console.WriteLine()`. Они работают только с одним строковым параметром для вывода сообщения и не допускают вывода значений за счет использования синтаксиса `{x}`. Это означает, что для включения численных значений в выводимую строку необходимо использовать оператор +. Однако существует возможность использовать второй строковый параметр (необязательный), который применяется для указания **категории** выводимого текста. Категория позволяет даже при беглом просмотре выходных сообщений в окне Output определять, с какими сообщениями мы имеем дело в тех случаях, когда похожие сообщения могут выводиться из разных мест программы.

Вывод этих функций имеет следующий вид:

<категория>: <сообщение>

Например, следующий оператор в качестве дополнительного параметра, определяющего категорию, использует параметр `"myFunc"`:

```
Debug.WriteLine("Added 1 to i", "myFunc");
```

Результат его выполнения будет следующий:

```
myFunc: Added 1 to i
```

Давайте рассмотрим пример.

Практикум: вывод текста в окно Output

1. Создайте новое консольное приложение с именем `Ch07Ex01` в директории `C:\BegCSharp\Chapter7`.

2. Внесите следующие изменения в программу:

```
using System;
using System.Diagnostics;

namespace Ch07Ex01
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      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);
         foreach (int index in maxValIndices)
         {
            Console.WriteLine(index);
         }
      }
```

```
        static int Maxima(int[] integers, out int[] indices)
        {
            Debug.WriteLine("Maximum value search started.");
            indices = new int[1];
            int maxVal = integers[0];
            indices[0] = 0;
            int count = 1;
            Debug.WriteLine("Maximum value initialized to " +
                            maxVal + ", at element index 0.");
            for (int i = 1; i < integers.Length; i++)
            {
                Debug.WriteLine("Now looking at element at index " + i + ".");
                if (integers[i] > maxVal)
                {
                    maxVal = integers[i];
                    count = 1;
                    indices = new int[1];
                    indices[0] = i;
                    Debug.WriteLine("New maximum found. New value is " + maxVal +
                                    ", at element index " + i + ".");
                }
                else
                {
                    if (integers[i] == maxVal)
                    {
                        count++;
                        int[] oldIndices = indices;
                        indices = new int[count];
                        oldIndices.CopyTo(indices, 0);
                        indices[count - 1] = i;
                        Debug.WriteLine("Duplicate maximum found at element index " +
                                        i + ".");
                    }
                }
            }
            Trace.WriteLine("Maximum value " + maxVal + " found, with " + count +
                            " occurrences.");
            Debug.WriteLine("Maximum value search completed.");
            return maxVal;
        }
    }
}
```

3. Запустите программу в отладочном режиме (см. рис. справа).



4. Завершите приложение и посмотрите содержимое окна Output (в режиме Debug):

```
'DefaultDomain': Loaded 'c:\winnt\microsoft.net\framework\v1.0.2901\mscorlib.dll',
No symbols loaded.
'Ch07Ex01': Loaded 'C:\BegCSharp\Chapter7\Ch07Ex01\bin\Debug\Ch07Ex01.exe',
Symbols loaded.
'Ch07Ex01.exe': Loaded
'c:\winnt\assembly\gac\system\1.0.2411.0__b77a5c561934e089\system.dll',
No symbols loaded.
'Ch07Ex01.exe': Loaded
'c:\winnt\assembly\gac\system.xml\1.0.2411.0__b77a5c561934e089\system.xml.dll',
No symbols loaded.
```

```
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 program '[2056] Ch07Ex01.exe' has exited with code 0 (0x0).
```

5. Перейдите в режим **Release**, воспользовавшись раскрывающимся меню панели инструментов **Standard** (стандартные) (см. рис. справа).



6. Повторно запустите программу, на этот раз в режиме **Release** (окончательный вариант), и снова после завершения выполнения приложения посмотрите окно **Output**:

```
'DefaultDomain': Loaded 'c:\winnt\microsoft.net\framework\v1.0.2901\mscorlib.dll',
No symbols loaded.
'Ch07Ex01': Loaded 'C:\BegCSharp\Chapter7\Ch07Ex01\bin\Release\Ch07Ex01.exe',
No symbols loaded.
'Ch07Ex01.exe': Loaded
'c:\winnt\assembly\gac\system\1.0.2411.0__b77a5c561934e089\system.dll', No
symbols loaded.
'Ch07Ex01.exe': Loaded
'c:\winnt\assembly\gac\system.xml\1.0.2411.0__b77a5c561934e089\system.xml.dll',
No symbols loaded.
Maximum value 9 found, with 2 occurrences.
The program '[1792] Ch07Ex01.exe' has exited with code 0 (0x0).
```

Как это работает

Это приложение является расширенной версией приложения, которое было использовано в предыдущей главе. В нем применяется функция для вычисления максимального значения, содержащегося в целом массиве. Этот вариант программы возвращает также массив, содержащий индексы элементов, значение которых равно максимальному, что позволяет вызывающему коду выполнять над этими элементами какие-либо действия. Давайте пройдем по этой программе.

Для начала обратите внимание, что в начале программы появился еще один оператор `using`:

```
using System.Diagnostics;
```

Он позволяет упростить доступ к функциям, которые обсуждались перед этим примером, поскольку все они содержатся в пространстве имен `System.Diagnostics`. Без этого оператора все строки кода вроде:

```
Debug.WriteLine("Bananas");
```

пришлось бы дополнительно квалифицировать, т. е. записывать в виде:

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

Оператор `using` упрощает программу и делает ее менее многословной.

Код, находящийся в функции `Main()`, просто инициализирует тестовый массив целых значений с именем `testArray`, а также объявляет еще один целый массив с именем `maxValIndices`, в котором будут храниться индексы, полученные от функции `Maxima()` (эта функция выполняет основные вычисления); затем код вызывает эту функцию и после окончания ее работы выводит полученные результаты.

Функция `Maxima()` является несколько более сложной, но в ней нет почти ничего такого, с чем вы еще не встречались. Поиск по массиву выполняется аналогичным способом, как и функция `MaxVal()` из предыдущей главы, за исключением того, что здесь ведется запись индексов элементов, обладающих максимальными значениями.

Возможно, ключевым кодом, на который следует обратить особое внимание (кроме строк, отвечающих за вывод отладочной информации), является метод, используемый для отслеживания индексов. Вместо того чтобы возвращать массив, который хранил бы все индексы (и, следовательно, был бы равен по размеру исходному массиву), функция `Maxima()` возвращает массив, чей размер соответствует количеству индексов, которое должно в нем храниться. Это достигается путем последовательного создания новых массивов по ходу поиска. Такой способ является необходимым, поскольку размеры уже созданного массива не могут изменяться.

В самом начале поиска делается предположение, что именно первый элемент исходного массива (для него используется локальное имя `integers`) имеет максимальное значение и что он единственный элемент. Следовательно, можно присвоить соответствующие значения переменной `maxVal` (она возвращает значение функции, равное найденному максимальному значению) и выходному массиву `indices`, в котором сохраняются индексы элементов с максимальным значением. Переменной `maxVal` присваивается значение первого элемента массива `integers`, а в `indices` заносится единственное значение — 0, которое является индексом первого элемента массива. Мы также сохраняем число обнаруженных максимальных значений в переменной с именем `count`, что позволяет отслеживать массив `indices`.

Основным телом функции является цикл, который проходит по всем значениям в массиве `integers`, за исключением самого первого, поскольку оно уже обработано. Очередное значение сравнивается с текущим значением переменной `maxVal` и игнорируется, если `maxVal` больше. Если проверяемое значение больше, чем `maxVal`, то в `maxVal` и `indices` вносятся соответствующие изменения. Если значение оказывается равным `maxVal`, то переменная `count` увеличивается на единицу, а для хранения индексов создается новый массив `indices`. Этот новый массив имеет размерность на единицу больше, чем предыдущий, и в него записывается новый обнаруженный индекс.

Код, реализующий последнюю функциональную возможность, имеет вид:

```
if (integers[i] == maxVal)
{
   count++;
   int[] oldIndices = indices;
   indices = new int[count];
   oldIndices.CopyTo(indices, 0);
   indices[count - 1] = i;
   Debug.WriteLine("Duplicate maximum found at element index " +
                   i + ".");
}
```

Здесь происходит дублирование старого массива индексов в `oldIndices`, который представляет собой целый массив, локальный для блока кода `if`. Обратите также внимание на то, что значения, хранящиеся в массиве `oldIndices`, копируются в новый массив `indices` с помощью функции *<имяМассива>*`.CopyTo()`. Эта функция получает в качестве параметров массив, в который осуществляется копирование, и индекс первого копируемого элемента, после чего переносит в этот массив все остальные значения.

С помощью функций `Debug.WriteLine()` и `Trace.WriteLine()` на всем протяжении программы выводятся различные текстовые сообщения. Таким способом в отладочном режиме мы получаем полный перечень выполняемых в цикле шагов, которые и приводят к результату. В рабочем режиме мы наблюдаем только итог вычислений, поскольку в этом режиме функция `Debug.WriteLine()` не работает.

Кроме рассмотренных функций с окончанием на `WriteLine()` существует несколько других функций, о которых также следует знать. Для начала упомянем функции, являющиеся эквивалентами функции `Console.Write()`:

- `Debug.Write()`
- `Trace.Write()`

Обе эти функции используют тот же синтаксис, что и функции `WriteLine()` (один или два параметра, первый — с текстом сообщения и второй — необязательный — с категорией), но при этом они не добавляют символы "конец строки".

Существуют и другие команды:

- `Debug.WriteLineIf()`
- `Trace.WriteLineIf()`
- `Debug.WriteIf()`
- `Trace.WriteIf()`

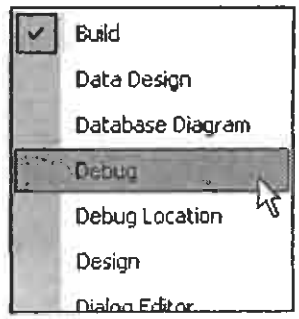
Каждая из этих функций имеет те же параметры, что и аналогичные функции "*без if*", с добавлением еще одного обязательного параметра, стоящего первым в списке параметров. В качестве этого параметра используется логическое значение (или выражение, в результате вычисления которого получается логическое значение), поэтому текст выводится только в том случае, если это значение `true`. Эти функции применяются для вывода текста в окно **Output** при выполнении некоторых условий. Например, мы можем наложить требование выводить отладочную информацию только в определенных ситуациях, т. е. в нашем коде может содержаться огромное количество обращений к функции `Debug.WriteLineIf()`, при этом выполнение каждой из них будет зависеть от текущих условий. Если соответствующее условие не выполняется, то вывода производиться не будет, соответственно, и окно **Output** не будет засоряться лишней информацией.

Отладка в режиме останова

Прочие способы отладки работают в режиме останова программы. Перейти в него можно несколькими способами, и все они тем или иным путем приводят к временной остановке выполнения программы. В данном разделе мы сначала познакомимся с этими способами, а затем рассмотрим вопрос, чего можно добиться переходом в такой режим.

Переход в режим останова

Простейшим способом перехода в режим останова является нажатие кнопки паузы в VS в процессе выполнения приложения. Эта кнопка располагается на панели инструментов **Debug**, которую необходимо добавить к тем панелям инструментов, которые появляются в VS по умолчанию. Для этого следует щелкнуть правой кнопкой мыши в области панели инструментов и выбрать панель **Debug** (см. рис. справа).

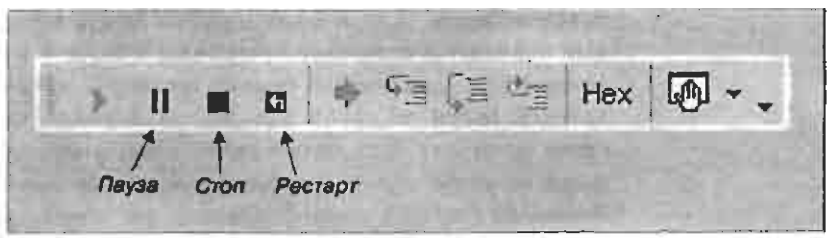




Появившаяся панель инструментов имеет вид, показанный на рисунке слева.

Первые четыре кнопки на этой панели предназначены для ручного управления остановкой программы. Три из них окрашены в серый цвет, так как они не могут работать с программой, которая не выполняется в настоящий момент. Доступная кнопка — "старт" — идентична кнопке, расположенной на стандартной панели инструментов.

После запуска приложения панель инструментов принимает вид, показанный на рисунке справа.



Теперь те три кнопки, которые прежде были серого цвета, перешли в рабочее состояние и позволяют:

- Временно прекратнть выполнение приложения (сделать паузу) и перейти в режим останова.
- Полностью прекратить выполнение приложения (это приводит не к переходу в режим останова, а к полному выходу из приложения).
- Заново запустить приложение.

Пауза, пожалуй, является наиболее простым способом перехода в режим останова, но она не дает возможности точно определить, в каком именно месте приложение будет остановлено. Наиболее распространено использование естественных остановок в работе приложения, например, когда приложение ожидает ввода

информации от пользователя. Существует также возможность временной остановки приложения при выполнении какой-либо продолжительной операции или длинного цикла, но в этом случае конкретная точка, в которой мы остановимся, окажется абсолютно случайной.

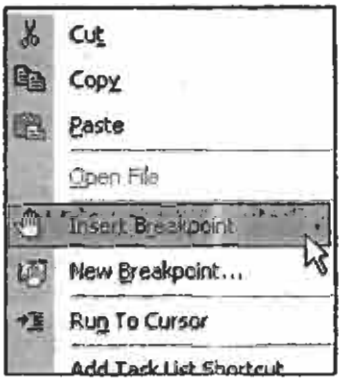
В общем случае гораздо полезнее использовать **точки останова** (breakpoints).

Точки останова

Точка останова — это некоторый маркер в исходном коде, который вызывает автоматическое переключение в режим останова. Возможны следующие настройки:

- Переходить в режим останова немедленно по достижении точки останова.
- Переходить в режим останова по достижении точки останова в том случае, если логическое выражение имеет значение `true`.
- Переходить в режим останова по достижении точки останова заданное число раз.
- Переходить в режим останова по достижении точки останова в том случае, если с момента предшествующего достижения этой точки значение переменной было изменено.

Обратите внимание, что все перечисленное выше оказывается возможным только в том случае, если приложение строилось в отладочном режиме. Если же откомпилировать приложение с построением окончательного варианта, то все точки останова будут проигнорированы.

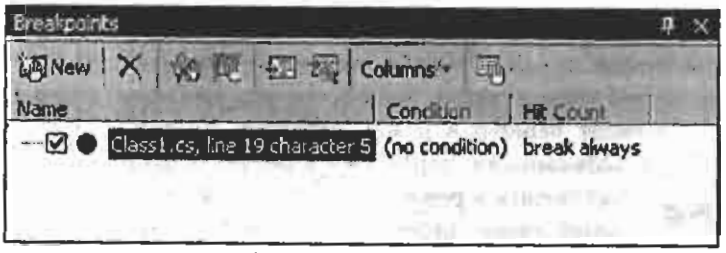


Существует три способа включать в программу точки останова. Для задания простых точек, которые приводят к остановке программы в момент их достижения, достаточно просто щелкнуть левой кнопкой мыши по серой области, расположенной слева от строки кода, или нажать правую кнопку мыши на самой строке и выбрать пункт меню **Insert Breakpoint** (вставка точки останова), как показано на рисунке справа.

Включение точки останова приведет к появлению красного кружка рядом с соответствующей строкой кода и к выделению самой строки:

```
11        static void Main(string[] args)
12        {
13            int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};
14            int[] maxValIndices;
15            int maxVal = Maxima(testArray, out maxValIndices);
16            Console.WriteLine("Maximum value (0) found at element indices:", maxVal);
17            foreach (int index in maxValIndices)
18            {
19                Console.WriteLine(index);
20            }
21        }
```

Мы можем также получить информацию обо всех точках останова в данном файле в окне **Breakpoints**. Необходимо активизировать это окно, выбрав пункт

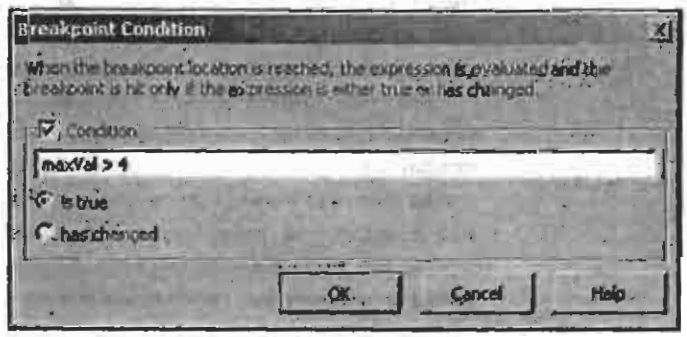


меню **Debug | Windows | Breakpoints**. В результате в нижней части экраиа появится окно (на том же месте, что и окна **Task List** и **Output**), показанное на рисунке слева.

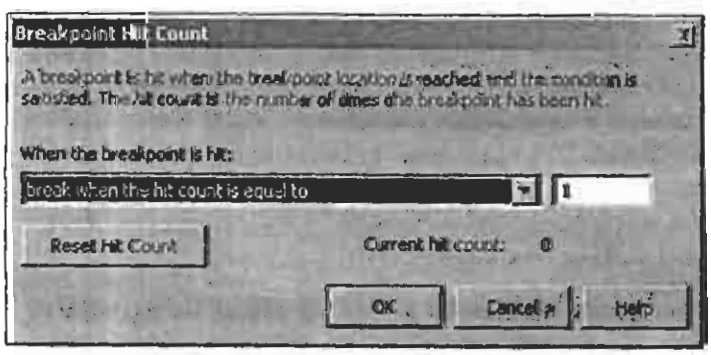
С помощью этого окна мы можем деактивировать точку останова (сняв флажок, расположеиный слева от описания точки останова; деактивированные точки изображаются в виде пустого красного кружка), убрать точку останова или изменить ее свойства.

Свойства, изображеиные в данном окне — **Condition** (условие) и **Hit Count** (счетчик повторений) — всего лишь часть имеющихся, однако они являются наиболее полезными. Мы можем отредактировать их, щелкнув правой киопкой мыши по соответствующей точке останова (в программе или в данном окне) и выбрав пункт меню **Properties**. Теперь мы получаем возможность использовать три закладки: **Function** (функция), **File** (файл) и **Address** (адрес) — для изменения местоположения точки останова (закладка **Address** позволяет устаиавливать контрольную точку по абсолютному адресу в памяти; это сложная тема, которой мы в данный момент касаться ие будем).

Нажатие кнопки **Condition** приводит к открытию диалогового окна, представленного на рисунке справа. В верхней части окна прочтем предупреждение: "Когда достигается точка останова, производится вычисление выражения, и останов сработает только в том случае, если значение выражения true или если это условие было изменено.", ниже следуют три флажка: "Условие" (выставив этот флажок, можно ввести вычисляемое выражение в расположенное под ним поле), "имеет значение true", "было изменено".



В этом окне можно задать произвольное логическое выражение, состоящее из любых переменных, область действия которых охватывает данную точку останова. На приведенном выше рисунке изображена точка останова, которая будет срабатывать, только если при ее достижении значение переменной `maxVal` больше 4. Мы можем также проверить, изменялось ли выражение, и остановиться, если это имело место (для вышеприведенного случая переход может быть осуществлен, только если значение `maxVal` изменилось с 2 на 6 с момента предыдущего достижения точки останова).

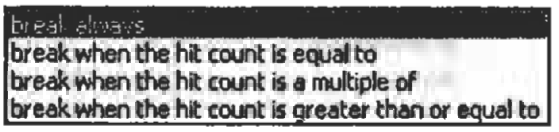


Нажатие кнопки **Hit Count** приводит к открытию диалогового окна, показанного на рисунке слева.

В верхней части окна прочтем предупреждение: "Точка останова срабатывает, когда она достигается и при этом условие оказывается выполненным. Счетчик повторений — это число срабатываний этой точки.", ниже из раскрывающегося списка можно выбрать условие: "Когда должна срабатывать

контрольная точка:". Под списком слева находится кнопка "Обнулить счетчик повторений", справа — информационная строка: "Текущее значение счетчика повторений:".

В этом диалоговом окне мы можем задать количество раз достижения точки останова, прежде чем произойдет переход в режим останова. Раскрывающийся список предлагает следующие возможности: переходить в режим останова всегда; переходить в режим останова, когда счетчик повторений равен; переходить в режим останова, когда счетчик повторений кратен; переходить в режим останова, когда счетчик повторений больше или равен (см. рис. слева).



Выбранная возможность совместно со значением, введенным в расположенное рядом со списком текстовое поле, и определяет поведение точки останова.

Счетчик повторений оказывается полезным для циклов с большим числом повторений, когда может потребоваться прервать выполнение программы, скажем, после первых 5000 проходов. Не имея такой возможности, мы бы настрадались, останавливая и запуская программу 5000 раз!

Другие способы перехода в режим останова

Существуют два других способа перехода в режим останова. Одни из них — это принятие решения о переходе в такой режим при возникновении **необрабатываемой исключительной ситуации**. Эта тема рассматривается ниже в данной главе. Другой способ прервать выполнение кода — создание **утверждения** (assertion).

Утверждения — это инструкции, позволяющие прерывать выполнение программы с выводом определяемого пользователем сообщения. Утверждения часто используются в процессе разработки приложения для проверки того, все ли идет правильно. Например, мы можем наложить требование, чтобы некоторая переменная в некоторой точке нашего приложения имела значение, меньшее 10. В этом случае мы можем воспользоваться утверждением для проверки выполнения требования и прерывания программы при его нарушении. При проверке можно использовать опцию Abort (окончить), которая завершит выполнение приложения; опцию Retry (повторить), которая приведет к останову; и опцию Ignore (игнорировать), которая позволит приложению выполняться дальше в обычном режиме.

Так же, как и функции для вывода отладочной информации, с которыми вы познакомились ранее, функция, позволяющая использовать утверждение, имеет две разновидности:

- `Debug.Assert()`
- `Trace.Assert()`

Отладочный вариант функции — `Debug.Assert()` — будет компилироваться только при создании отладочной версии.

Эти функции имеют три параметра. Первый из них — логическое значение, причем если оно равно `false`, то срабатывает утверждение. Второй и третий параметры имеют строковое значение и записывают информацию как в появляющееся окно диалога, так и в окно Output. Для приведенного выше примера потребуется следующий вызов функции:

```
Debug.Assert(myVar < 10, "Значение переменной myVar больше или равно 10.",
                         "Невыполнение утверждения в Main().");
```

Утверждения часто оказываются полезными на ранних этапах использования приложения пользователями. Мы можем распространить рабочую версию прило-

жения с функциями `Trace.Assert()`, что позволит разработчику отслеживать корректность функционирования данного приложения. Если какое-либо утверждение окажется невыполненным, пользователь сообщит об этом разработчикам, которые таким образом получают возможность проанализировать проблему, несмотря на то, что они остаются в неведении, каким образом произошел сбой.

Можно, например, привести краткое описание ошибки в первой строке и инструкции, объясняющие, что необходимо предпринять, во второй строке:

```
Trace.Assert(myVar < 10, "Выход значения переменной за допустимые границы.",
                         "Пожалуйста, сообщите разработчику код ошибки KCW001.");
```



В том случае, если это утверждение окажется невыполненным, пользователь увидит на экране сообщение: "**Выход значения переменной за допустимые границы. Пожалуйста, сообщите разработчику код ошибки KCW001.**", ниже: "**в Class1.Main(String[] args)**"

Если у пользователя инсталлирован VS и он нажмет кнопку **Retry** для рабочей версии, то он увидит не исходный код, а инструкции приложения на языке ассемблера, которые он вряд ли сможет понять. Ниже приводится участок кода на языке ассемблера для примера из предыдущего раздела:

```
00000196 nop
00000197 pop      ebx
00000198 pop      esi
00000199 pop      edi
0000019a mov      esp,ebp
0000019c pop      ebp
0000019d ret      4
```

В этом совсем не просто разобраться, и только те программисты, у которых имеется опыт работы с языком ассемблера, могут на что-то рассчитывать. А это значит, что наша программа будет скрыта от (большинства) любопытных глаз!

Следующие темы касаются возможностей, появляющихся при останове приложения. Вообще говоря, мы будем переходить в режим останова для того, чтобы обнаружить ошибку в программе (или убедиться в том, что все работает правильно). При останове можно использовать самые разнообразные методы, каждый из которых позволяет производить анализ кода и конкретного состояния приложения в точке временного прекращения работы.

Мониторинг содержимого переменных

Мониторинг содержимого переменных представляет собой пример той помощи, которую оказывает нам VS. Самый простой способ узнать значение, хранящееся в переменной, это подвести указатель мыши к имени переменной в исходном коде, находясь в режиме останова. В этом случае на экране появится вспомогательное поле желтого цвета, в котором содержится информация об этой переменной, включая ее текущее значение.

Мы можем выделять таким же образом целые выражения, чтобы получить информацию о результате их выполнения. Однако применение этого способа весьма ограниченно, поскольку он не позволяет, например, просматривать содержимое массивов.

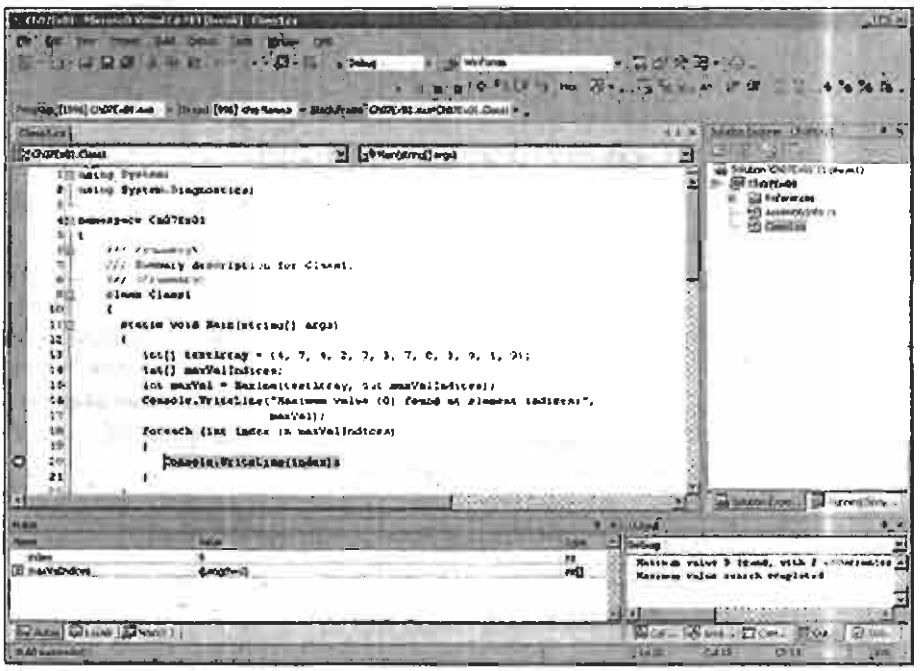
Вы, вероятно, уже успели заметить, что когда мы запускаем приложения посредством VS, то внешнее представление различных окон меняется. Во время выполнения программы по умолчанию происходит следующее:

- Исчезает окно **Properties**.
- К окну **Solution Explorer** добавляется вкладка **Running Documents** (используемые документы), которая показывает использующиеся в проекте документы, если таковые существуют.
- Размер окна **Output** изменяется, поскольку половину нижней части экрана занимает новое окно.

Это новое окно оказывается очень полезным для целей отладки. Оно позволяет отслеживать значения переменных в приложении, которое находится в режиме останова. В нем имеются три вкладки, которые используются следующим образом:

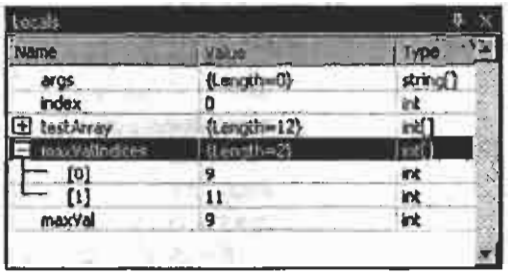
- **Autos** отображает переменные, используемые в текущем и предшествующем операторах.
- **Locals** отображает все действующие переменные.
- **Watch** *N* предлагает настраиваемый вывод переменных и выражений, где *N* изменяется от 1 до 4.

Новый вид экрана представлен на следующем рисунке:



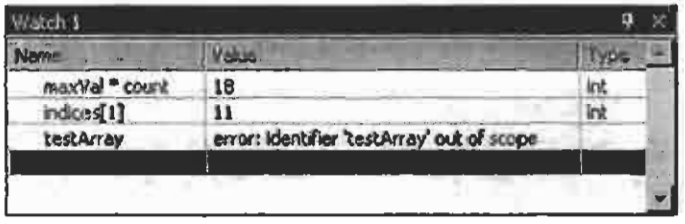
Все эти окна работают более-менее аналогичным образом, предоставляя различные дополнительные возможности в зависимости от их конкретного предназначения. Общим для всех окон является наличие списка переменных, в котором представлена информация об имени, значении и типе переменной. Более сложные переменные, такие как массивы, могут просматриваться посредством дополнительного запроса, для чего используются символы развертывания/свертывания (+ и –)

с левой стороны от каждого имени, позволяющие представлять содержимое переменных в виде дерева. Например, на рисунке справа изображен вид экрана, который был получен посредством включения точки останова в код предыдущего примера непосредственно после вызова функции `Maxima()`. Здесь для просмотра развернута одна из переменных типа массива — `maxValIndices`.



В этом окне также имеется возможность редактировать содержимое переменных, что позволяет эффективно обходить все другие способы присваивания, которые могли встречаться в предшествующем коде. Для этого достаточно просто ввести новое значение в столбце Value (значение) для той переменной, которую мы хотим отредактировать. Этот способ может использоваться, к примеру, в тех случаях, когда требуется проверить работу некоторых сценариев, для чего в противном случае пришлось бы вносить изменения в программу.

Окна Watch (просмотр), которых может быть не более 4, позволяют осуществлять мониторинг определенных переменных или выражений, содержащих определенные переменные. Для того чтобы воспользоваться этим окном, достаточно просто набрать имя переменной или выражения в столбце Name (имя) и посмотреть на результат. Не следует забывать, что некоторые переменные в приложении в данный момент могут находиться вне области своего действия и будут соответствующим образом помечены в окне Watch. На рисунке слева изображено окно просмотра с некоторыми переменными и выражениями, использованными в качестве примера. В данном случае вновь использован код из предыдущей программы, выполнение которой остановлено на функции `Maxima()`.



Массив `testArray` является локальным по отношению к `Main()`, поэтому в таком случае мы не можем увидеть его значений. Вместо этого мы получаем сообщение о том, что данная переменная находится вне своей области действия.

Существует также возможность добавлять переменные в окно Watch, перенося их мышью из исходного кода.

Воспользоваться остальными окнами можно с помощью пунктов меню Debug | Windows | Watch | Watch *N*. В каждом окне может содержаться свой собственный набор переменных и выражений, что позволяет группировать связанные между собой переменные для удобства просмотра.

Помимо всех этих окон просмотра существует еще окно QuickWatch (быстрый просмотр), которое позволяет оперативно получать информацию о переменных из исходного кода. Для этого следует щелкнуть правой кнопкой мыши по интересующей вас переменной и выбрать опцию меню QuickWatch. В большинстве случаев, однако, это не намного проще использования обычных окон Watch.

Следует обратить внимание на один важный момент, касающийся окон просмотра: эти окна сохраняются между запусками приложения. Если мы окончим некоторое приложение, а затем запустим его повторно, то вновь создавать окна просмотра не придется — VS запомнит, что нас интересовало в предыдущий раз.

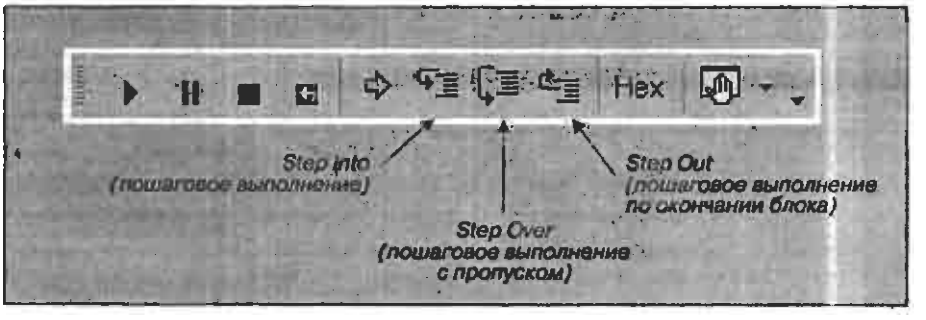
Пошаговое выполнение кода

Итак, мы рассмотрели способы, позволяющие узнать, что происходит в приложении в той точке, в которой оно переходит в режим останова. Теперь предстоит познакомиться с использованием VS для пошагового выполнения кода из режима останова, что позволяет сразу в точности узнавать результаты выполнения этого кода. Этот способ оказывается незаменимым для тех из нас, кто не в состоянии думать со скоростью компьютера.

Когда мы переходим в режим останова, то слева от текста программы появляется стрелка (если для перехода в режим останова использовалась точка останова, то эта стрелка может изначально появляться внутри красного кружка, обозначающего точку останова), которая указывает на ту строку кода, которая должна выполняться:

```
11        static void Main(string[] args)
12        {
13            int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};
14            int[] maxValIndices;
15            int maxVal = Maxima(testArray, out maxValIndices);
16            Console.WriteLine("Maximum value (0) found at element indices:", maxVal);
17            foreach (int index in maxValIndices)
18            {
19                Console.WriteLine(index);
20            }
21        }
```

Стрелка показывает нам, какой точки программы мы достигли на момент перехода в режим останова. Отсюда мы можем продолжить выполнение программы в пошаговом режиме. Для этого нам потребуются некоторые другие кнопки, расположенные на уже знакомой нам панели инструментов Debug:



Шестая, седьмая и восьмая кнопки позволяют управлять ходом выполнения программы в режиме останова. В порядке следования они означают:

- **Step Into** — выполняется очередной оператор, перед выполнением следующего делается остановка.
- **Step Over** — аналогично предыдущему, но пропускаются вложенные блоки кода.
- **Step Out** — текущий блок кода выполняется до конца, после чего перед очередным оператором восстанавливается режим останова.

Если возникает необходимость просматривать все выполняемые приложением отдельные операции, то для последовательного выполнения каждой инструкции можно воспользоваться режимом **Step Into**. Этот режим предполагает и пошаговое выполнение встретившихся функций, таких как `Maxima()` из предыдущего примера.

Если этот режим будет выбран в момент, когда стрелка достигнет строки 15, в которой осуществляется вызов функции `Maxima()`, то стрелка перейдет на первую строку кода внутри этой функции. Напротив, если при достижении строки 15 мы выберем режим **Step Over** с пропуском, то стрелка сразу переместится на строку 16 и не будет проходить по строкам кода функции `Maxima()` (хотя этот код все равно будет выполнен). Если, выполняя программу, мы попали в функцию, выполнение которой нас не интересует, то существует возможность установить режим **Step** и вернуться в код, который вызвал данную функцию.

В процессе пошагового выполнения кода значения переменных, вероятнее всего, будут изменяться. Наблюдая за окнами просмотра, которые описывались в предыдущем разделе, мы сможем увидеть все происходящие изменения.

Если в программе присутствуют семантические ошибки, этот метод является одним из самых полезных. Выполняя программу в пошаговом режиме, мы получаем возможность достигнуть той точки кода, в которой предположительно могут содержаться ошибки, и эти ошибки проявятся точно так же, как если бы программа выполнялась в обычном режиме. Параллельно можно осуществлять контроль за данными и видеть, что именно работает неправильно. Позднее в настоящей главе мы воспользуемся этим способом, чтобы разобраться, что происходит в использованном в качестве примера приложении.

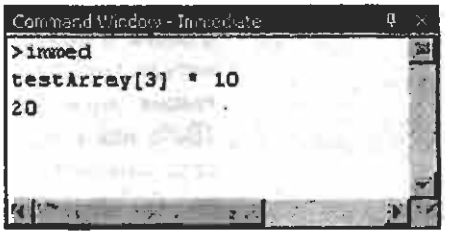
Нам осталось рассмотреть еще два окна. В процессе отладки в окне **Task List / Output** появляются еще две вкладки: **Command Window** (командное окно) и **Call Stack** (стек вызовов).

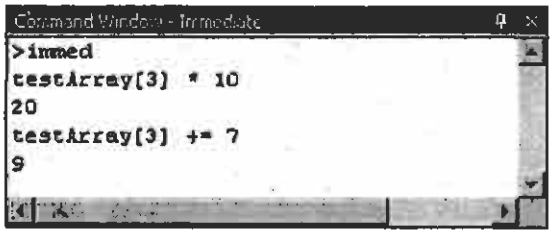
Немедленное выполнение команд

Окно **Command Window** имеет два режима: **Command** (командный) и **Immediate** (немедленного выполнения). В режиме **Command** операции VS выполняются без использования меню и т. п., а режим **Immediate** позволяет выполнять код в промежутках между выполнением обычных строк кода и вычислять значения выражений. В режиме **Command** в начале каждой строки появляется символ "больше" (>). Мы можем перейти в режим немедленного выполнения команд, набрав **"immed"** и нажав *Enter*, а вернуться назад можно, набрав **">cmd"** и нажав *Enter*.

Этот раздел будет посвящен в основном режиму **Immediate**, поскольку режим **Command** по-настоящему полезен только при выполнении сложных операций.

Самый простой вариант использования этого окна — для вычисления выражений, что в некотором смысле напоминает "одноразовое" использование окон **Watch**. Для этого достаточно просто ввести выражение и нажать *Enter*. Запрашиваемая информация немедленно будет выведена на экран (см. рис. справа).





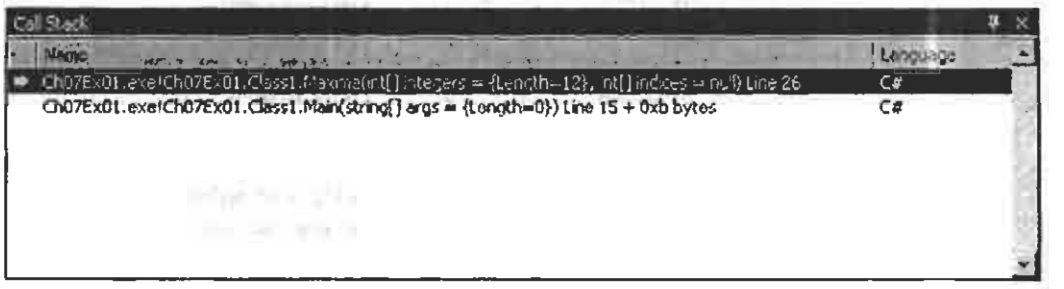
Здесь же можно внести необходимые изменения в содержимое переменной (см. рис. слева).

В большинстве случаев добиться желаемых результатов можно более простым способом, а именно с помощью окон мониторинга переменных, однако данный способ может оказаться удобным для просмотра значений. Пригодится он и для проверки тех выражений, результаты которых, скорее всего, впоследствии не будут нас интересовать.

Окно стека вызовов

В последнем окне, которое мы рассмотрим в этом разделе, показывается путь, по которому мы пришли в данную точку. Проще говоря, мы можем увидеть в нем список вложенных вызовов функций. Здесь также хранится конкретная точка, в которой этот вызов был осуществлен.

В примере, использовавшемся выше, в том случае, если бы мы перешли в режим останова, находясь внутри функции `Maxima()`, или попали бы в эту функцию в результате пошагового выполнения кода, мы бы увидели следующее окно:



Это окно особенно полезно в тех случаях, когда ошибка уже обнаружена, поскольку оно позволяет увидеть, что происходило непосредственно перед ее возникновением. Если ошибка находится в часто используемых функциях, то это помогает определить ее причину.

> *Необходимо учитывать, что в этом окне может оказаться совершенно непонятная информация. Иногда ошибки происходят вне нашего приложения, например, в результате неправильного использования внешних функций. В таких случаях в этом окне может оказаться длинный список вхождений, из которых только одно или два окажутся вам знакомы.*

Обработка ошибок

Первая часть данной главы была посвящена способам поиска и исправления ошибок в процессе разработки приложения, направленным на то, чтобы исключить их появление в окончательном варианте программы. Существуют, однако, ситуации, когда заранее известно, что ошибки могут возникать, и способа добиться 100%-ной уверенности в их отсутствии не существует. В таких случаях рекомендуется заранее предусмотреть возникновение проблем и написать такой код, который мог бы аккуратно с ними справиться без прекращения выполнения приложения.

Обработка ошибок — это общее название всех способов такого рода; в данном разделе будут рассмотрены возникновение исключительных ситуаций (exceptions) и методы работы с ними.

Исключительные ситуации

Исключительная ситуация — это ошибка, возникшая либо в нашем коде, либо в вызванной нашим кодом функции, которая проявилась в период выполнения. В данном случае термин "ошибка" имеет более широкое значение, чем до сих пор, поскольку исключительные ситуации могут создаваться в функциях и т. п. вручную.

Например, мы можем создать исключительную ситуацию в функции в том случае, если один из ее строковых параметров не начинается с символа "a". Строго говоря, вне контекста настоящей функции это ошибкой не является, хотя с точки зрения кода, вызывающего данную функцию, это, безусловно, ошибка.

Исключительные ситуации встречались нам несколько раз на протяжении этой книги. Возможно, самым простым примером является попытка обратиться к элементу массива, выходящему за его границы:

```
int[] myArray = {1, 2, 3, 4};
int myElem = myArray[4];
```

В результате появится приведенное ниже сообщение о исключительной ситуации, а выполнение приложения будет прекращено:

An unhandled exception of type 'System.IndexOutOfRangeException' occurred in *<file>*.exe
(Необрабатываемая исключительная ситуация типа
'System.IndexOutOfRangeException', возникшая в *<файл>*.exe)

Здесь *<файл>* — это имя файла, в котором содержится данная исключительная ситуация.

Исключительные ситуации описаны в пространствах имен, и большинство из них обладает именами, из которых становится ясно, для чего они предназначены. В приведенном примере возникшая исключительная ситуация имеет название `System.IndexOutOfRangeException` ("исключительная ситуация: выход индекса из диапазона допустимых значений"), что совершенно верно, поскольку мы задали индекс, который не входит в диапазон допустимых значений для массива `myArray`.

Появление сообщения и завершение выполнения приложения происходит только в тех случаях, когда исключительная ситуация является необрабатываемой. А что требуется сделать для того, чтобы можно было "обработать" исключительную ситуацию?

try...catch...finally

В языке программирования C# предусмотрен специальный синтаксис для **Structured Exception Handling (SEH — структурированной обработки исключительных ситуаций)**. Существуют специальные ключевые слова, позволяющие помечать участки кода как участки, в которых допустима обработка исключительных ситуаций, и размещать инструкции, указывающие, что следует делать при возникновении исключительной ситуации. Для этих целей используются три ключевых слова: `try`, `catch` и `finally`. Каждому из них соответствует некоторый блок кода, и все они должны помещаться в последовательно расположенных строках кода. Основная конструкция имеет следующий вид:

```
try
{
   . . .
}
catch (<типИсключительнойСитуации> e)
{
   . . .
}
finally
{
   . . .
}
```

Существует также возможность использовать блоки `try` и `finally` без блока `catch`, а также блок `try` с несколькими блоками `catch`. Если имеется один или несколько блоков `catch`, то блок `finally` является необязательным, в противном случае он обязателен.

Эти блоки используются следующим образом:

- `try` содержит код, который **может привести** к возникновенню исключительной ситуации.
- `catch` содержит код, который должен выполняться в случае возникиовення исключительной ситуацин. Блоки `catch` могут быть настроеиы таким образом, чтобы реагнровать только на один конкретный тип исключительной ситуацни (например, на `System.IndexOutOfRangeException`), для чего используется параметр *<типИсключительнойСитуации>*, поскольку существует возможность одновременно установить несколько блоков `catch`. Но можно и вообще опустить этот параметр, тогда получится **общий** блок `catch`, который будет реагнровать на все виды исключительных ситуаций.
- `finally` содержит блок кода, который выполняется всегда: либо после выполнения блока `try`, если не возникло ннкаких исключительных ситуаций, либо после блока `catch`, если производилась обработка исключительной снтуации, либо непосредственно перед тем, как приложение будет завершено из-за возникновения иеобрабатываемой исключительной снтуации (последнее и объясняет необходимость этого блока; в протнвном случае мы могли бы с тем же успехом разместить этот код просто вслед за блоком).

События, происходящие после возникновения исключительной ситуации в блоке кода `try`, имеют такую последовательность:

- Выполнение блока `try` прекращается в той точке, в которой возннкла исключительная ситуация.
- Если существует блок `catch`, то производится проверка на предмет того, совпадает ли его параметр с типом возникшей исключительной ситуации. Если блок `catch` отсутствует, то выполняется блок `finally` (который в этом случае является обязательным).
- Если блок `catch` существует, но значение его параметра не совпадает с типом возникшей исключительной ситуации, проверка производится в других блоках `catch`.
- Если параметр блока `catch` совпадает с типом возникшей исключительной ситуации, то сначала выполняется код, содержащийся в блоке, а затем — в случае, если он прнсутствует,— выполняется блок `finally`.
- Если ни один из блоков `catch` не соответствует типу возникшей исключительной ситуации, то выполняется блок `finally` (если он существует).

Давайте рассмотрим пример, демонстрнрующий обработку исключительных ситуаций.

Практикум: запись текста в окно Output

1. Создайте новое консольное приложение с именем `Ch07Ex02` в директории `C:\BegCSharp\Chapter7`.

2. Внесите следующие изменения в программу:

```
class Class1
{
   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.");                    // Line 18
            Console.WriteLine("ThrowException(\"{0}\") called.", eType);
                                                                               // Line 19
            ThrowException(eType);
            Console.WriteLine("Main() try block continues.");                  // Line 21
         }
         catch (System.IndexOutOfRangeException e)                             // Line 23
         {
            Console.WriteLine("Main() System.IndexOutOfRangeException catch"
                              + " block reached. Message:\n\"{0}\"", e.Message);
         }
         catch                                                                 // Line 29
         {
            Console.WriteLine("Main() general catch block reached.");
         }
         finally
         {
            Console.WriteLine("Main() finally block reached.");
         }
         Console.WriteLine();
      }
   }

   static void ThrowException(string exceptionType)
   {
      Console.WriteLine("ThrowException(\"{0}\") reached.", exceptionType);
                                                                               // Line 43
      switch (exceptionType)
      {
         case "none" :
            Console.WriteLine("Not throwing an exception.");
            break;                                                             // Line 48
         case "simple" :
            Console.WriteLine("Throwing System.Exception.");
            throw (new System.Exception());                                    // Line 51
            break;
         case "index" :
            Console.WriteLine("Throwing System.IndexOutOfRangeException.");
            eTypes[4] = "error";                                               // Line 55
            break;
         case "nested index" :
            try                                                                // Line 58
            {
               Console.WriteLine("ThrowException(\"nested index\") " +
                                 "try block reached.");
               Console.WriteLine("ThrowException(\"index\") called.");
               ThrowException("index");                                        // Line 63
            }
```

```
                catch                                                    // Line 65
                {
                   Console.WriteLine("ThrowException(\"nested index\") general"
                                     + " catch block reached.");
                }
                finally
                {
                   Console.WriteLine("ThrowException(\"nested index\") finally"
                                     + " block reached.");
                }
                break;
             }
          }
       }
```



3. Запустите приложение (см. рис. слева).

Как это работает

У этого приложения имеется блок `try` в функции `Main()`, в котором вызывается фуикция `ThrowException()`. Названная функция может создавать различные исключительные ситуации в зависимости от того, какой параметр используется при обращении к ней:

- `ThrowException("none")` — не вызывает возникновения исключительной ситуации.
- `ThrowException("simple")` — вызывает возникновение общей исключительной ситуации.
- `ThrowException("index")` — создает исключительную ситуацию `System.IndexOutOfRangeException`.
- `ThrowException("nested index")` — содержит свой собственный блок `try`, в котором располагается код, вызывающий `ThrowException("index")` с целью создания исключительной ситуации `System.IndexOutOfRangeException`.

Все эти параметры обладают типом `string` и хранятся в глобальном массиве `eTypes`, который в функции `Main()` проходится в цикле — для того, чтобы функция `ThrowException()` была вызвана по одному разу с каждым возможным значением параметра. В процессе выполнения этих итераций на консоль выводятся поясняющие сообщения.

Этот пример предоставляет прекрасную возможность использовать метод пошагового выполнения программы, с которым вы уже познакомились. Проходя по всей программе строка за строкой, вы будете в точности понимать, каким образом они выполняются.

Добавьте новую точку останова (со свойствами по умолчанию) в 18-ю строку кода, которая выглядит следующим образом:

```
Console.WriteLine("Main() try block reached.");
```

Обратите внимание, что номера строк используются в том виде, в каком они появляются в загружаемой версии программы. Если вывод номеров строк отключен, в него можно войти с помощью пункта меню Tools | Options... и раздела Text Editor | C# | General. Поскольку в вышеприведенный код включены комментарии, вы можете пройти текст, не имея перед собой открытого файла.

Запустите приложение в отладочном режиме.

Программа почти мгновенно перейдет в режим останова, при этом стрелка будет находится на 18-й строке. Если вы выберете вкладку **Locals** окна мониторинга переменных, то увидите, что текущее значение `eType` равняется `"none"`. Воспользуйтесь кнопкой **Step Into** для выполнения строк 18 и 19 и убедитесь в том, что первая строка текста выведена на консоль. Далее используйте кнопку **Step Into** для того, чтобы войти в функцию `ThrowException()` на строке 20.

Внутри фуикции `ThrowException()` (на строке 43) окно **Locals** претерпевает определенные изменения. Теперь мы находимся вне области действия переменных `eType` и `args` (поскольку по отношению к `Main()` они являются локальными) и вместо них видим аргумент `exceptionType`, значение которого равняется, естественно, `"none"`. Продолжайте нажимать **Step Into**, и вы доберетесь до оператора `switch`, в котором проверяется значение переменной `exceptionType` и выполняется код, выводящий на экран строку **"Not throwing an exception"**. После того как будет выполнен оператор `break` на строке 48, мы выйдем из функции `ThrowException()` и продолжим выполнение функции `Main()` со строки 21. Поскольку никакой исключительной ситуации не возникло, выполнение блока `try` будет продолжено.

Далее работа функции продолжается выполнением блока `finally`. Нажмите **Step Into** еще несколько раз, чтобы завершить выполнение блока `finally` и первую итерацию цикла `foreach`. Когда мы в следующий раз попадаем на строку 20, то вызываем функцию `ThrowException()` уже с другим параметром — `"simple"`.

Продолжайте нажимать **Step Into** для дальнейшего продвижения по функции `ThrowException()`, и в конце концов вы доберетесь до строки 51:

```
throw(new System.Exception());
```

В данном случае, для создания исключительной ситуации используем ключевое слово `throw`. В качестве параметра при нем стоит вновь (`new`) инициализируемая исключительная ситуация. В данном случае мы используем еще один тип исключительных ситуаций из пространства имен `System` — `System.Exception`.

Выполнение этого оператора с помощью **Step Into** приводит к переходу в общий блок `catch`, который начинается со строки 29. Поскольку в предшествующем блоке `catch`, начинающемся со строки 23, совпадения не обнаружены, то вместо него выполняется данный блок `catch`. Пошаговое выполнение позволяет пройти по всем операторам этого блока, затем по операторам блока `finally` и вернуться на очередную итерацию цикла `foreach`, в котором на строке 20 функция `ThrowException()` вызывается с очередным параметром. На этот раз параметр принимает значение `"index"`.

В этом случае функция `ThrowException()` создает исключительную ситуацию в строке 55:

```
eTypes[4] = "error";
```

Массив `eTypes` является глобальным, поэтому возможность доступа к нему существует. Однако в этой строке кода делается попытка обратиться к пятому элементу

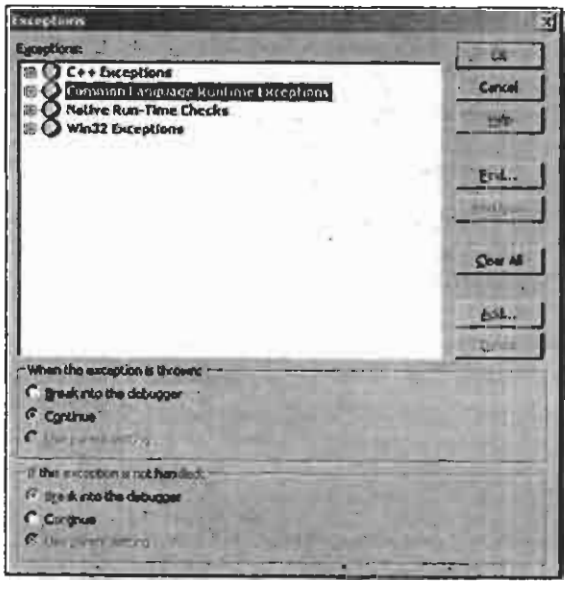
массива (не забудьте, что нумерация в массивах начинается с нуля), что приводит к возникновению исключительной ситуации `System.IndexOutOfRangeException`.

На этот раз в функции `Main()` предусмотрен соответствующий блок `catch`, и пошаговое выполнение программы приводит нас в начало настоящего блока на строку 23.

Вызов `Console.WriteLine()` в этом блоке выводит сообщение, хранящееся в описании исключительной ситуации посредством `e.Message` (мы обладаем доступом к описанию исключительной ситуации через параметр блока `catch`). Дальнейшее пошаговое выполнение программы через блок `finally` (но не через второй блок `catch`, поскольку исключительная ситуация уже обработана) снова приводит нас к выполнению очередной итерации цикла и к очередному вызову функции `ThrowException()` на строке 20.

Когда мы достигаем конструкции `switch` в функции `ThrowException()`, то входим в новый блок `try`, который начинается со строки 58. При попадании на строку 63, происходит выполнение вложенного вызова функции `ThrowException()` — на этот раз с параметром `"index"`. Если хотите, можете воспользоваться кнопкой **Step Over** для того, чтобы пропустить выполняемые здесь строки кода, поскольку мы их уже проходили. Как и в предыдущий раз, этот вызов создает исключительную ситуацию `System.IndexOutOfRangeException`. Однако теперь исключительная ситуация обрабатывается во вложенной конструкции `try...catch...finally`, которая находится внутри функции `ThrowException()`. В этой структуре отсутствует явное совпадение с данным типом исключительной ситуации, поэтому его обработкой занимается общий блок `catch` (начинающийся со строки 65).

Как и при обработке исключительных ситуаций в предыдущих случаях, мы проходим в пошаговом режиме этот блок и соответствующий ему блок `finally` и достигаем конца вызванной функции. Однако в данном случае имеется одно принципиальное отличие: хотя исключительная ситуация и была создана, она также была и обработана — кодом, находящимся в функции `ThrowException()`. Это означает, что для обработки в функции `Main()` исключительных ситуаций не осталось, поэтому мы напрямую попадаем в блок `finally`, после чего приложение завершает свою работу.



Просмотр списков и настройка исключительных ситуаций

.NET Framework содержит огромное количество типов исключительных ситуаций, и мы обладаем полной свободой создавать и обрабатывать эти исключительные ситуации в своих программах и даже создавать исключительные ситуации для обработки более сложными приложениями. VS предоставляет специальный диалог (см. рис. слева), позволяющий просматривать и редактировать имеющиеся исключительные ситуации, который можно вызвать через пункт меню **Debug | Exceptions...** (или нажатием комбинации клавиш *Ctrl+Alt+E*).

Исключительные ситуации выводятся в соответствии со своей категорией и пространством имен библиотеки .NET. Исключительные ситуации, включенные в пространство имен `System`, можно просмотреть, развернув сначала элемент Common

Language Runtime Exceptions, а затем элемент System. В открывшемся списке можно обнаружить исключительную ситуацию `System.IndexOutOfRangeException`, которую мы использовали ранее.

Любая исключительная ситуация может быть настроена с помощью переключателей, расположенных в нижней части окна Exceptions. Большинство исключительных ситуаций по умолчанию имеют установку Use parent setting (использовать установку более высокого уровня); это означает, что для них применяются опции уровня всей категории (они полностью совпадают с изображенными на вышеприведенном рисунке). Первая группа переключателей, When the exception is thrown (когда создалась исключительная ситуация), может использоваться для перехода в отладчик даже в том случае, если исключительная ситуация уже обработана. Вторая группа позволяет игнорировать необрабатываемые исключительные ситуации независимо от того, к каким последствиям это может привести.

В большинстве случаев установки по умолчанию нас вполне устраивают.

Замечания по поводу обработки исключительных ситуаций

Обратите внимание на то, что блоки `catch`, предназначенные для более специфической обработки исключительных ситуаций, необходимо располагать перед блоками, предназначенными для более общей обработки. Если это требование не будет выполнено, то приложение не пройдет компиляцию.

Также необходимо помнить, что мы можем создавать исключительные ситуации непосредственно из блоков `catch` либо способом, использованным в предыдущем примере, либо просто с помощью выражения:

```
throw;
```

Это выражение приведет к тому, что прерывание, уже обработанное блоком `catch`, будет сгенерировано повторно.

Если исключительная ситуация будет создана с помощью `throw`, то она будет обрабатываться не текущей конструкцией `try...catch...finally`, а конструкцией более высокого уровня (хотя блок `finally` вложенной конструкции все равно выполнится).

Если, например, внести следующие изменения в блок `try...catch...finally`, расположенный в функции `ThrowException()`:

```
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 выполнение этой программы сначала продолжится выполнением блока `finally`, изображенного здесь, а затем блока `Main()`, в котором будет обнаружено совпадение. В результате выходной поток изменится следующим образом:



На этом рисунке можно заметить дополнительные строки, выведенные из функции `Main()`, поскольку исключительная ситуация `System.IndexOutOfRangeException` перехватывается именно в этой функции.

Итоги

Эта глава посвящена преимущественно способам, позволяющим производить отладку приложений. Таких способов несколько, и большинство из них доступно независимо от типа создаваемого приложения.

Были рассмотрены следующие темы:

- Использование функций `Debug.WriteLine()` и `Trace.WriteLine()` для вывода текста в окно Output.
- Режим останова и способы перехода в него, включая изменяемые точки останова.
- Окна в VS, содержащие отладочную информацию.
- Пошаговое выполнение программ.
- Обработка исключительных ситуаций с помощью конструкции `try...catch...finally`.

Теперь вы познакомились со всем, что может потребоваться для создания простых консольных приложений, включая методы их отладки. В следующей части этой книги мы перейдем к рассмотрению объектно-ориентированного программирования.

Упражнения

1. "Использование функции `Trace.WriteLine()` более предпочтительно, чем использование функции `Debug.WriteLine()`, поскольку второй вариант работает только при построении отладочной версии". Согласны ли вы с этим утверждением? Почему?

2. Напишите программу для простого приложения, в котором содержится цикл, генерирующий ошибку после 5000 итераций. Используйте точку останова для перехода в режим останова непосредственно перед возникновением ошибки на 5000-й итерации. (*Подсказка*: наиболее простой способ сгенерировать ошибку — это попытаться получить доступ к несуществующему элементу массива, например, к `myArray[1000]` для массива, содержащего 100 элементов.)

3. "Блоки `finally` выполняются только в том случае, если не выполняется блок `catch`". Верно ли данное утверждение?

4. Воспользовавшись перечислимым типом данных `orientation`, описанным ранее, напишите приложение, которое использует Structured Exception Handling (SEH — структурированную обработку исключительных ситуаций) для перевода переменной типа `byte` в переменную типа `orientation` безопасным способом. Обратите внимание, что вы можете форсировать возникновение исключительных ситуаций с помощью ключевого слова `checked`, пример использования которого приводится ниже. Эту часть кода вы должны использовать в своем приложении:

```
enum orientation : byte
{
   north = 1,
   south = 2,
   east  = 3,
   west  = 4
}
```

```
myDirection = checked((orientation)myByte);
```



Введение в объектно-ориентированное программирование

В предыдущих главах мы рассмотрели синтаксис языка C# и основы программирования на нем. Кроме того, мы научились собирать работоспособные консольные приложения. Однако для того, чтобы получить доступ ко всему потенциалу C# и .NET Framework, необходимо научиться использовать методы **объектно-ориентированного программирования (ООП)**. На самом деле некоторые из этих методов уже применялись нами, однако чтобы не усложнять жизнь, мы не акцентировали на этом внимание.

В этой главе мы на некоторое время отвлечемся от кода и сосредоточимся на принципах, лежащих в основе ООП. Такой подход очень скоро приведет нас обратно к языку C#, поскольку последний неразрывно связан с ООП. Ко всем понятиям, введенным в настоящей главе, мы будем возвращаться в дальнейшем, иллюстрируя их с помощью примеров программ, так что не следует впадать в панику, если не удается сразу понять материал.

Для начала рассмотрим базовые понятия ООП, которые и ответят на самый фундаментальный вопрос: "Что такое объект?". Очень скоро вы увидите, что существует весьма большое количество терминов, связанных с ООП; поначалу это может показаться достаточно сложным, хотя используемый язык будет подробно разъяснять. Вы также убедитесь, что применение ООП требует нового взгляда на программирование.

Помимо обсуждения общих принципов, лежащих в основе ООП, мы рассмотрим одну область, для которой исчерпывающее понимание ООП является необходимым: приложения Windows Forms. Этот тип приложений (т. е. приложения, в которых используется среда Windows и такие возможности этой среды, как меню, кнопки и т. п.) оставляет широкий простор для описания объектов, и мы постараемся эффективно проиллюстрировать принципы ООП в среде Windows Forms.

> *Обратите внимание, что ООП в том виде, в котором оно представлено в настоящей главе, на самом деле является ООП для .NET, и поэтому некоторые из способов, обсуждаемых здесь, неприменимы к другим разновидностям среды ООП. Такой подход при изучении C# является вполне оправданным, поскольку впоследствии это существенно упростит вам жизнь.*

Итак, давайте начнем с самого начала.

Что такое объектно-ориентированное программирование

Объектно-ориентированное программирование — это относительно новый подход к созданию компьютерных приложений, в котором предпринимается попытка решить многие проблемы, возникающие при применении "традиционного" программирования. Тип программирования, с которым вы были до сих пор знакомы, известен под названием **функционального** (или **процедурного**) программирования. Он очень часто приводит к созданию так называемых **монолитных** приложений, в которых все функциональные возможности содержатся в небольшом числе модулей (зачастую в одном). Применяя методы ООП, мы очень часто будем использовать большое количество программных модулей, каждый из которых предоставляет только одну функциональную возможность, причем каждый модуль может существовать отдельно от остальных и даже быть совершенно независимым. Такой модульный метод написания программ оказывается более гибким и увеличивает возможности для повторного использования кода.

В качестве более наглядной иллюстрации ООП представим себе сложное приложение в виде высококлассного гоночного автомобиля. Если бы использовались традиционные способы программирования, то этот автомобиль представлял бы собой единое целое. И пожелай мы его усовершенствовать, нам бы пришлось заменить его целиком, отправив на завод-изготовитель для доводки заводскими специалистами (или купить новый). Если же применяются методы ООП, то можно просто приобрести новый двигатель и, следуя прилагающимся к нему инструкциям, заменить его самостоятельно.

В "традиционном" приложении порядок выполнения прост и прямолинеен. Приложения загружаются в память, их выполнение начинается в точке А и завершается в точке В, после чего они выгружаются из памяти. В процессе выполнения такого приложения могут использоваться самые разнообразные объекты, например, файлы на различных носителях или возможности, предоставляемые видеокартой, однако основное тело вычислений расположено в едином месте. Весь код, как правило, предназначается для манипуляций над данными с помощью различных математических и логических средств. Способы такого манипулирования обычно очень просты благодаря применению базовых типов (таких как целый и логический) для более сложного представления данных.

При использовании ООП такая прямолинейность встречается редко. И хотя результат достигается тот же самый, способ его получения зачастую оказывается совершенно отличным. Методы ООП в большей степени основываются на структуре и значении данных, а также на взаимодействии между различными данными. Это обычно требует больших усилий при разработке проекта, но дает преимущество расширяемости. После того как соглашение о способе представления конкретного типа данных достигнуто, оно может использоваться и в более поздних версиях приложения, и в совершенно новых приложениях. Тот факт, что такое соглашение уже существует, позволяет радикально сократить время, затрачиваемое на разработку. Это позволяет объяснить вышеприведенный пример с гоночным автомобилем. В этом случае в качестве соглашения применяется такая структура кода для "двигателя", которая позволяет использовать новый код (новый двигатель) взамен старого без всяких проблем и не требует отправки автомобиля на завод.

Кроме соглашений о представлении данных, ООП часто позволяет упростить жизнь за счет соглашений о структуре и об использовании более абстрактных сущностей. Например, соглашение может быть достигнуто не только относительно формата данных, который должен применяться при отправке выходного потока на какое-либо устройство вроде принтера, но и относительно способов обмена данными с этим устройством. Это будет касаться инструкций, которые оно может воспринимать и в дальнейшей работе.

Как можно предположить по названию, все это достигается за счет использования некоторых **объектов**. Так что же собой представляет объект?

Что такое объект

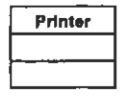
Объект — это строительный блок ООП-приложения. В этом блоке инкапсулируется некоторая часть приложения, которая может представлять собой процесс, группу данных или какую-либо более сложную сущность.

В простейшем смысле объект очень напоминает тип структуры, с которым вы познакомились раньше и в котором содержатся члены в виде переменных и функций различных типов. Совокупность переменных представляет данные, хранящиеся в этом объекте, а функции обеспечивают доступ к функциональным возможностям объекта. Несколько более сложные объекты могут вообще не содержать никаких данных; вместо это они могут представлять некоторый процесс и состоять исключительно из функций. Например, можно использовать объект, представляющий принтер, которой состоит из функций, обеспечивающих контроль за работой этого устройства (распечатать документ, распечатать тестовую страницу и т. д.).

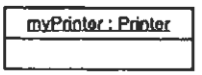
Объекты в C# создаются на основе типов точно так же, как и переменные. Тип объекта известен в ООП под специальным названием **"класс объекта"**. Мы можем использовать определения классов для создания **экземпляров** объектов, что означает создание реального поименованного **экземпляра** данного класса. Словосочетание "экземпляр данного класса" и термин "объект" в данном случае означают одно и то же, однако уверяем вас, что, вообще говоря, "класс" и "объект" обозначают фундаментально отличающиеся понятия.

В данной главе мы будем описывать классы и объекты с использованием синтаксиса **Unified Modeling Language (UML — унифицированный язык моделирования)**. UML — это язык, разработанный для моделирования приложений с использованием тех объектов, из которых они строятся, тех операций, которые они выполняют, и тех вариантов их применения, для которых они предназначаются. В данном случае мы будем использовать только основы этого языка, объясняя требующиеся нам понятия по ходу дела и не отвлекаясь на более сложные аспекты.

> *Диаграммы, использованные в данной главе, создавались с помощью программы Microsoft Visio, которая поставляется с коммерческой версией VS.*



На рисунке справа приводится UML-представление класса, предназначенного для описания принтера, названного `Printer`. Имя класса изображено в верхней секции этого квадрата (к двум его нижним секциям мы обратимся немного позднее).



На рисунке слева изображено UML-представление экземпляра класса `Printer` с именем `myPrinter`. Здесь в верхней секции располагается имя экземпляра, за ним следует имя соответствующего класса. Имена разделяются двоеточием.

Свойства и поля

Свойства и поля обеспечивают доступ к данным, хранящимся в объекте. Данные, содержащиеся в объектах,— это то, что отличает один объект от другого, поскольку в различных объектах одного и того же класса могут **храниться** различные значения свойств и полей.

Теперь нам потребуется ввести еще один термин. Различные части информации, хранящиеся в объекте, в совокупности определяют состояние объекта.

Представьте себе класс объектов с именем `CupOfCoffee` (чашка кофе), описывающий чашку кофе. При создании экземпляра этого класса (т. е. при создании объекта данного класса) мы должны описать его состояние, для того чтобы объект был осмысленным. В этом случае мы могли бы использовать такие свойства и поля, которые позволят программе, использующей этот объект, задавать марку кофе, определять, добавлены ли в кофе сахар и молоко, является ли этот кофе растворимым и т. д. В таком случае объект, представляющий чашку кофе, будет обладать определенным состоянием, например, "колумбийский фильтрованный кофе с молоком и двумя кусочками сахара".

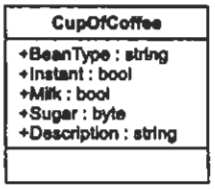
Как поля, так и свойства являются типизированными, поэтому информацию, содержащуюся в них, можно хранить в переменных типа `string`, `int` и т. п. Однако свойства отличаются от полей тем, что они не обеспечивают *непосредственного* доступа к данным. Объекты обладают возможностью изолировать пользователей, которым не требуется точное представление о структуре существующих свойств, от реального устройства своих данных. Если для описания числа кусочков сахара в экземпляре `CupOfCoffee` мы используем поле, то пользователи смогут занести в это поле любое значение, какое им заблагорассудится; если же мы используем свойство, то мы сможем ограничить это значение диапазоном от 0 до 2.

Вообще говоря, для организации доступа к состоянию лучше использовать свойства, а не поля, поскольку в этом случае мы обладаем большим контролем за происходящим, хотя синтаксис и в том, и в другом случае применяется один и тот же. Режим доступа к свойствам также может быть четко определен для данного объекта. Некоторые свойства могут использоваться в режиме "только чтение", что дает возможность просматривать их, но не изменять (по крайней мере, непосредственно). Очень часто полезными оказываются способы, позволяющие одновременно считывать несколько различных частей информации. Например, мы можем описать свойство класса `CupOfCoffee` с именем `Description` (описание), использующееся в режиме "только чтение", которое при обращении будет возвращать строку, описывающую состояние некоторого экземпляра данного класса (например, строку, приведенную выше). Эти же самые данные можно было бы получить, обращаясь к нескольким различным свойствам, однако использование одного свойства позволяет экономить время и усилия. Аналогичным образом можно использовать свойства, работающие в режиме "только запись".

Подобно тому как мы имеем возможность установить режим чтения/записи свойств, мы можем задать тип доступа совместно для полей и свойств. Это позволяет определять, какие программы имеют доступ к этим членам: доступны ли они для всего кода (public — **общие**), только для кода в рамках данного класса (private — **частные**), или же доступ к ним определяется по более сложной схеме (этот вопрос будет рассматриваться в настоящей главе по мере необходимости). Чрезвычайно распространенной является практика, когда поля описываются как частные, а доступ к ним организовывается посредством свойств, описанных как общие. Это означает, что код в рамках данного класса обладает непосредственным доступом к данным, хранящимся в поле, а общее свойство скрывает эти данные от

внешних пользователей и не позволяет им записывать в эти данные недопустимые значения. Принято говорить, что общие члены предоставляются данным классом.

Для большей наглядности давайте отождествим это с областью действия переменной. Частные поля и свойства в таком случае можно представить как локальные по отношению к объекту, которому они принадлежат, в то время как область действия общих полей и свойств также распространяется и на код, внешний по отношению к данному объекту.



При представлении класса посредством UML вторая секция используется для изображения свойств и полей.

На рисунке слева можно видеть представление нашего класса `CupofCoffee`, в котором описано пять членов (неважно, свойств или полей — в UML между ними не существует никаких отличий), обсуждавшихся нами ранее. Каждое вхождение содержит следующую информацию:

- Тип доступа: символ + используется для общего члена, символ – используется для частного члена. В дальнейшем мы не будем изображать частные члены на диаграммах данной главы, поскольку эта информация является внутренней по отношению к классу. Не будет приводится также и информация, касающаяся режима доступа (чтение/запись).
- Имя члена.
- Тип члена.

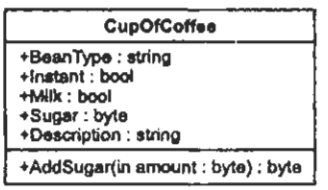
В качестве разделителя между именем и типом члена используется двоеточие.

Методы

Метод — это термин, используемый для обозначения функций, предоставляемых данным объектом. Методы могут вызываться точно так же, как и любые другие функции, и в них так же, как и в функциях, могут использоваться возвращаемые значения и параметры (более подробно эта тема описывается в главе 6).

Методы применяются для обеспечения доступа к функциональным возможностям объектов. Подобно полям и свойствам, они могут быть общими или частными, ограничивая по мере необходимости доступ к ним из внешнего кода. Методы часто учитывают состояние объекта при своей работе и обладают доступом к частным членам, например, к частным полям. Так, в классе `CupofCoffee` мы можем описать метод с именем `AddSugar()` (добавить сахар), который обеспечит использование более удобного синтаксиса для увеличения значения свойства сладости кофе, чем присвоение соответствующего значения свойству `Sugar` (сахар).

В UML при описании объектов для изображения функций используется третья секция прямоугольника (см. рис. справа).



Используемый в этом случае синтаксис аналогичен применяемому для полей и свойств, за исключением того, что в конце строки указывается тип возвращаемого значения, а также приводятся параметры. В UML каждый параметр изображается с одним из следующих идентификаторов: In, out или inout. Эти идентификаторы используются для обозначения направления потока данных, при этом out и inout в первом приближении соответствуют применяемым в C# ключевым словам `out` и `ref`, описанным в главе 6. Идентификатор in примерно соответствует такому поведению C#, когда отсутствуют оба ключевых слова.

Объектом может быть все

Настал момент, когда необходимо, наконец, объясниться. Мы использовали объекты, свойства и методы на всем протяжении этой книги. В действительности в C# и .NET Framework объектом может быть все, что угодно. Функция `Main()` в консольном приложении является методом класса. Все типы переменных, рассмотренных нами, являются классами. Каждая из использовавшихся команд представляет собой свойство или метод, например, `<Строка>.Length`, `<Строка>.ToUpper()` и т. д. В данном случае точка отделяет имя экземпляра объекта от свойства или от имени метода.

Объекты существуют буквально повсеместно, и синтаксис, который требуется для их использования, зачастую оказывается очень простым. Действительно, он настолько прост, что до настоящего времени позволял нам сосредоточиться на более фундаментальных аспектах C#.

С этого момента мы приступаем к более детальному рассмотрению объектов. Имейте в виду, что понятия, которые будут вводиться, обладают весьма широкой областью использования. Они применимы даже к той простой маленькой переменной типа `int`, с которой вы так любили играться. К счастью, мы можем с успехом использовать их и для более сложных понятий.

Жизненный цикл объекта

У каждого объекта имеется четко определенный жизненный цикл, который длится с момента использования определения класса до уничтожения объекта. Кроме обычного состояния "используется", в жизненном цикле объекта присутствуют еще два важных этапа:

- **Создание объекта** — состояние, когда происходит первоначальное создание экземпляра объекта. Такая инициализация известна под названием создания объекта и осуществляется **конструктором** объекта.
- **Уничтожение объекта** — состояние, когда происходит уничтожение объекта, при котором очень часто возникает необходимость проведения различных восстановительных мероприятий, например освобождения памяти. Эта работа входит в функции деструктора объекта.

Конструкторы

Инициализация объекта происходит автоматически. Нам не приходится беспокоиться, допустим, о поиске свободной памяти, в которой будет размещен новый объект. Однако иногда возникают ситуации, когда на этапе инициализации может потребоваться выполнение каких-либо дополнительных действий. Например, очень часто бывает необходимо инициализировать данные, хранящиеся в объекте. Как раз это и входит в функции конструктора.

У каждого объекта имеется **конструктор, используемый по умолчанию**, который представляет собой метод без параметров. Его имя совпадает с именем самого класса. Определение класса может включать в себя несколько конструкторов. Они могут иметь отличающиеся сигнатуры, которые используются для создания экземпляра объекта. Для присваивания начальных значений данным, хранящимся внутри объекта, часто применяются конструкторы с параметрами.

В C# конструктор можно вызвать, введя ключевое слово new. Например, мы можем создать объект типа string следующим образом:

```
string myString = new string();
```

Объекты также могут создаваться с помощью **конструктора, используемого не по умолчанию**. Как и имя конструктора по умолчанию, имена этих конструкторов совпадают с именем класса, но у них имеются еще и параметры. Используются они аналогичным образом:

```
string myString = new string('a', 10);
```

В результате применения этого конструктора будет создана новая строка, которой в качестве начального значения будет присвоено "aaaaaaaaaa", т. е. символ "a", повторенный десятикратно.

Конструкторы, подобно полям, свойствам и методам, могут быть общими или частными. Код, внешний по отношению к данному классу, не может использовать для создания экземпляра объекта частный конструктор; для этого необходимо воспользоваться общим конструктором. Таким способом мы можем, например, заставить пользователей нашего класса применять конструктор, использующийся не по умолчанию.

В некоторых классах вообще отсутствуют общедоступные конструкторы, что означает невозможность для внешнего кода создавать экземпляры данного класса. Однако, как вы вскоре сможете убедиться, такие классы вовсе не являются совершенно бесполезными.

Деструкторы

Деструкторы используются в .NET Framework для того, чтобы выполнять уборку за объектами. В общем случае нам не требуется писать какой бы то ни было код для метода деструктора; напротив, за нас будет работать операция, выполняющаяся по умолчанию. Однако если необходимо выполнить какие-либо важные действия перед уничтожением экземпляра объекта, то можно задать определенные инструкции.

Очень важно запомнить, что метод деструктора объекта не вызывается сразу, как только данный объект перестает использоваться.

Когда, например, мы выходим за область действия переменной, она становится недоступной для нашего кода, но при этом она может по-прежнему существовать где-нибудь в памяти компьютера. Только после того как сборщик мусора среды исполнения .NET выполнит свою работу, она будет окончательно уничтожена.

Это означает, что не следует полагаться на деструктор в плане освобождения ресурсов, которые использовались экземпляром объекта, поскольку объект может оставаться неиспользуемым на протяжении длительного времени. Если используемые ресурсы критичны, то это может привести к возникновению проблем. Однако здесь имеется решение, и оно рассматривается в разделе "Удаляемые объекты" этой главы.

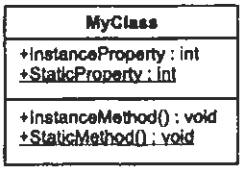
Статические члены класса и члены класса экземпляра

Наряду с такими членами, как свойства, методы и поля, относящимися к конкретным экземплярам объектов, существует возможность использовать статические (static) члены (известные также под названием членов с общим доступом (shared)), среди которых могут быть методы, свойства и поля. Доступ к статическим членам имеют все экземпляры данного класса, поэтому их можно представить себе в качестве глобальных для всех объектов данного класса. Статические свойства и поля позволяют осуществлять доступ к данным независимо от экземпляров объектов, а статические методы позволяют выполнять команды, которые относятся к данному типу объектов, но не к конкретным экземплярам объектов. Для того чтобы использовать статические члены нам, по существу, даже не требуется создавать экземпляры объектов с типом данного класса.

Использовавшиеся нами методы `Console.WriteLine()` и `Convert.ToString()` являются статическими. Ни в какой момент времени от нас не требуется создавать экземпляры классов `Console` или `Convert` (более того, даже если бы мы и попытались это сделать, то все равно ничего бы не получилось, поскольку к конструкторам этих классов не существует общего доступа, что обсуждалось ранее).

Существуют различные ситуации, при которых статические свойства и методы могут быть использованы с большой пользой.

В синтаксисе UML статические члены классов выделяются подчеркиванием (см. рис. справа).



Методы ООП

Теперь, когда вы познакомились с основами и узнали, что представляют собой объекты и как они работают, мы потратим определенное время на изучение некоторых других возможностей объектов. В этом разделе будут рассматриваться:

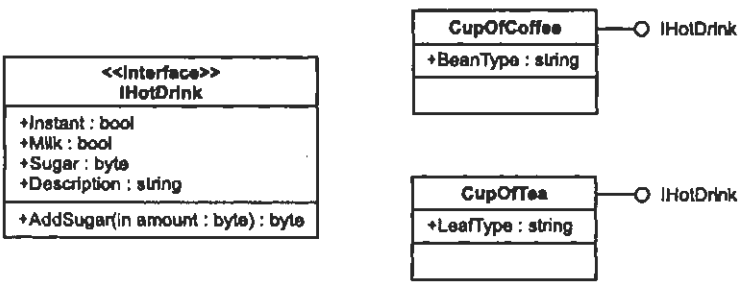
- Интерфейсы
- Наследование
- Полиморфизм
- Взаимоотношения между объектами
- Перегрузка операторов
- События

Интерфейсы

Интерфейсом называется семейство явно описанных как public методов и свойств, которые сгруппированы в единое целое и инкапсулируют какую-либо определенную функциональную возможность. После того как интерфейс определен, его можно реализовать в некотором классе. Это означает, что класс будет поддерживать все свойства и члены, определяемые данным интерфейсом.

Обратите внимание, что интерфейсы не могут существовать сами по себе. Мы не можем "создать экземпляр интерфейса" таким же образом, как мы создаем экземпляр класса. Кроме того, интерфейс не может содержать в себе никакого кода, который бы реализовал его члены; он просто описывает эти члены. Их реализация должна находиться в классах, в которых реализован данный интерфейс.

В предыдущем примере, касающемся кофе, мы могли бы сгруппировать в интерфейс несколько свойств и методов наиболее общего назначения, например, `AddSugar()`, `Milk` (молоко), `Sugar` (сахар) и `Instant` (растворимый). Такой интерфейс можно назвать как-нибудь вроде `IHotDrink` (горячий напиток) (имена интерфейсов обычно предваряются заглавной буквой `I`). Этот интерфейс допустимо использовать и для каких-нибудь других объектов, возможно, принадлежащих к классу `CupOfTea` (чашка чая). Другими словами, мы получаем возможность обращаться с этими объектами одинаковым образом, а они при этом по-прежнему могут обладать своими собственными индивидуальными свойствами (например, `BeanType` (сорт кофейных бобов) для `CupOfCoffee` и `LeafType` (сорт чайного листа) для `CupOfTea`).



Интерфейсы, реализованные на объектах UML, изображаются с помощью "леденцового" (lollipop) синтаксиса. На диаграмме, приведенной на рисунке слева, члены `IHotDrink` выделены в отдельный прямоугольник с помощью синтаксиса, напоминающего используемый для описания классов (к сожалению, существующая версия Visio не позволяет интерфейсам иметь поля или свойства).

У класса может быть несколько интерфейсов, и несколько классов могут поддерживать один и тот же интерфейс. Понятие интерфейса позволяет, таким образом, облегчить жизнь другим разработчикам и пользователям. Допустим, что у вас имеется некий код, в котором задействован объект, обладающий определенным интерфейсом. Если предположить, что вы не используете другие свойства и методы данного объекта, то появляется возможность заменять один объект на другой (код, в котором используется интерфейс `IHotDrink`, может работать как с экземпляром класса `CupOfCoffee`, так и с экземпляром класса `CupOfTea`). Кроме того, разработчик самого объекта имеет возможность снабжать вас усовершенствованными версиями этого объекта — если он поддерживает использовавшийся ранее интерфейс, то не составит никакого труда применить новую версию объекта в вашей программе.

Удаляемые объекты

Одним из интерфейсов, представляющих особенный интерес, является интерфейс `IDisposable`. На объекте, который поддерживает интерфейс `IDisposable`, должен быть реализован метод `Dispose()`, т. е. в нем должен содержаться код для реализации этого метода. Этот метод может вызываться, когда объект больше не нужен (например, перед выходом из области его действия); он освобождает некие критические ресурсы, которые в противном случае будут удерживаться до тех пор, пока сборкой мусора не будет запущен метод деструктора. Это дает дополнительные возможности для управления ресурсами, которые используются объектами.

В C# имеется возможность применять специальную конструкцию, которая замечательно использует этот метод. Ключевое слово `using` позволяет нам инициализировать некий объект, использующий критические ресурсы, внутри некоторого блока кода, в конце которого автоматически вызывается метод `Dispose()`. Вот каким образом можно реализовать такую конструкцию:

```
using (<ИмяКласса> <ИмяПеременной> = new <ИмяКласса>())
{
   . . .
}
```

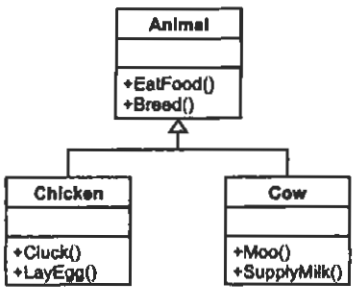
В этой конструкции переменную *<ИмяПеременной>* можно использовать по всему блоку, но в конце блока она будет автоматически удалена (посредством вызова метода `Dispose()`).

Наследование

Наследование является одной из наиболее важных особенностей ООП. Любой класс может **наследоваться** от другого класса, что означает, что он будет обладать всеми членами того класса, от которого он **наследуется**. В терминологии ООП класс, от которого происходит наследование (или от которого происходит **наследующий** класс), называется **классом-родителем** (иначе известным под названием **базового** класса). В C# все объекты происходят от единственного базового класса.

Наследование позволяет расширять или создавать более специфические классы на основе одного базового класса. Например, представьте себе класс, описывающий скотный двор (такой класс использовался асом-разработчиком восьмидесятилетним Стариной Макдональдом в его приложении для работы с домашними животными). Этот класс можно назвать `Animal` (животное) и включить в него такие методы, как `EatFood()` (принимать пищу) и `Breed()` (размножаться). После этого мы можем создать производный класс с именем `Cow` (корова), который будет поддерживать все эти методы, но при этом может обладать и своими собственными, например, `Moo()` (мычать) и `SupplyMilk()` (давать молоко). Мы можем создавать и другие производные классы, например, класс `Chicken` (курица) с методами `Cluck()` (кудахтать) и `LayEgg()` (нести яйца).

В UML наследование изображается с помощью стрелок (см. рис. справа).



Здесь для краткости опущены члены, обладающие возвращаемыми значениями.

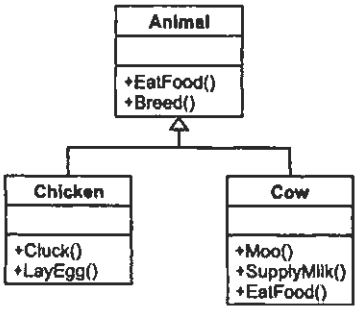
При наследовании от одного базового класса возникает очень важный вопрос о режиме доступа к членам данного класса. Частные члены базового класса, в отличие от общих членов, доступны не будут. Однако общие члены будут доступны как производному классу, так и внешнему коду. Это означает, что при использовании только этих двух режимов доступа мы не можем получить член, который был бы доступен базовому классу и производному классу, но не был бы при этом доступен для внешнего кода.

Для решения этой проблемы вводится третий режим доступа — protected (**защищенный**), при котором доступ открыт только для производных классов. С точки зрения внешнего кода этот режим доступа идентичен режиму доступа к частному члену — ни к одному, ни к другому доступа не имеется.

Кроме степени защиты данного члена мы можем определять его поведение при наследовании. Члены базового класса могут описываться как virtual (**виртуальный**), что означает, что такой член может быть переопределен классом, который его наследует. Другими словами, производный класс может использовать альтернативную реализацию данного члена. Такая альтернативная реализация не отменяет

оригинальный код, который по-прежнему остается доступным в рамках этого класса, однако он становится скрытым для внешнего кода. Если же никакой альтернативной реализации не используется, то внешний код обладает доступом к реализации данного члена в базовом классе.

Заметьте, что виртуальные члены не могут быть частными, поскольку это привело бы к парадоксу: нельзя одновременно утверждать, что член может быть переопределен производным классом и в то же время быть для него недоступным.

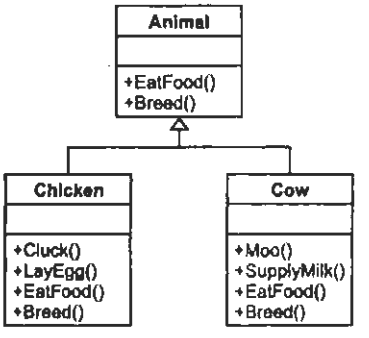


В примере с животными мы можем объявить метод `EatFood()` виртуальным и использовать для него другую реализацию во всех производных классах, например, в классе `Cow` (см. рис. слева).

Здесь метод `EatFood()` изображен как для класса `Animal`, так и для класса `Cow` с той целью, чтобы показать, что в каждом классе используется своя собственная реализация.

Базовые классы могут описываться как abstract (**абстрактные**). Абстрактный класс не допускает непосредственного создания экземпляра. Для того чтобы можно было им воспользоваться, он должен быть унаследован другим классом. У абстрактных классов могут иметься абстрактные члены, не имеющие никакой реализации, т. е. их реализация *должна* присутствовать в производном классе.

Если допустить, что класс `Animal` — абстрактный, то в UML он должен изображаться так, как показано на рисунке справа.



Имена абстрактных классов выводятся курсивом (или иногда прямоугольник, в котором они находятся, ограничивается пунктирной линией).

Наконец, класс может быть описан как sealed (**изолированный**). Изолированный класс нельзя использовать в качестве базового класса, поэтому у него не может быть производных классов.

В C# существует общий базовый класс для всех объектов, называемый `object` (этот же класс в .NET Framework называется `System.Object`). В следующей главе мы познакомимся с ним поближе.

Интерфейсы, описанные в этой главе, также могут наследоваться от других интерфейсов. Однако в отличие от классов интерфейсы могут наследоваться от нескольких базовых интерфейсов (таким же образом, как классы могут поддерживать по несколько интерфейсов).

Полиморфизм

Одно из следствий использования иерархии наследования заключается в том, что методы и свойства, предоставляемые производными классами, перекрываются. По этой причине часто существует возможность применять объекты, являющиеся экземплярами классов, в качестве объектов базового типа и использовать для них общий идентичный синтаксис. Например, если базовый класс с именем `Animal` обладает методом `EatFood()`, то синтаксис вызова этого метода из производных классов `Cow` и `Chicken` будет одинаковым:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

Полиморфизм позволяет пойти еще дальше. Существует возможность присвоить переменной базового типа переменную, которая имеет один из производных типов. Например:

```
Animal myAnimal = myCow;
```

При этом не требуется никакого приведения типов. С помощью этой переменной мы получаем возможность обращаться к методам базового класса:

```
myAnimal.EatFood();
```

Это приведет к вызову метода `EatFood()`, реализованного в производном классе. Обратите внимание, что не существует возможности вызывать аналогичным способом методы, определенные в производном классе. Следующий код работать не будет:

```
myAnimal.Moo();
```

Однако мы можем привести тип переменной базового типа к типу производного класса и после этого вызвать метод производного класса:

```
Cow myNewCow = (Cow)myAnimal;
myNewCow.Moo();
```

Такое приведение типа послужит причиной возникновения исключительной ситуации в том случае, если исходная переменная не будет относиться к классу `Cow` или классу, производному от класса `Cow`. Существуют способы указать, к какому типу принадлежит объект, но этот вопрос будет рассмотрен в следующей главе.

Полиморфизм оказывается чрезвычайно полезным методом, позволяя выполнять операции над различными объектами, происходящими от одного класса, с использованием минимального количества кода.

Обратите внимание, что использовать полиморфизм могут не только классы, у которых имеется один и тот же непосредственный класс-родитель: существует возможность таким же образом обращаться, например, к дочернему и внучатому классам, поскольку у них имеется общий класс в иерархии наследования.

И еще одно замечание. Запомните, что в C# все классы являются производными от базового класса `object`, который является корневым в их иерархии наследования. Следовательно, с любыми объектами можно обращаться как с экземплярами класса `object`. Именно поэтому функция `Console.WriteLine()` оказывается в состоянии обрабатывать бесконечное количество комбинаций параметров при формировании строки. Каждый параметр, кроме первого, рассматривается как экземпляр класса `object`, в результате чего возможен вывод на экран выходного потока произвольного объекта. Для этого вызывается метод `ToString()` (член класса `object`). Мы можем переопределить этот метод посредством некоторой реализации, подходящей для нашего класса, либо просто использовать вариант по умолчанию, который возвращает имя класса (квалифицированное в соответствии с тем пространством имен, в котором оно находится).

Полиморфизм интерфейсов

Ранее мы уже ввели понятие интерфейса, который позволяет сгруппировать взаимосвязанные методы и свойства. Хотя создавать экземпляры интерфейсов подобно тому, как создаются экземпляры объектов, нельзя, мы можем использовать переменные интерфейсного типа. В этом случае появляется возможность доступа к методам и свойствам, предоставляемым этим интерфейсом, на объектах, которые его поддерживают.

Предположим, что вместо того, чтобы для описания метода EatFood() использовать базовый класс Animal, мы включили метод EatFood() в интерфейс с именем IConsume. Оба класса — и Cow, и Chicken — могут поддерживать этот интерфейс; единственное отличие в данном случае заключается в том, что они обязательно должны иметь некоторую реализацию EatFood() (поскольку интерфейсы сами по себе не содержат никакой реализации). Тогда мы можем получить доступ к этому методу с помощью следующего кода:

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface = myCow;
consumeInterface.EatFood();
consumeInterface = myChicken;
consumeInterface.EatFood();
```

Это простой способ, позволяющий вызывать несколько различных объектов одинаковым образом, не опираясь на общий базовый класс. В приведенном выше отрывке кода осуществляется вызов метода consumInterface.EatFood() либо класса Cow, либо класса Chicken — в зависимости от того, какой из экземпляров был присвоен переменной интерфейсного типа.

Взаимоотношения между объектами

Наследование — это простой тип взаимоотношений между объектами, при котором производный класс полностью наследует свойства базового класса, причем производный класс может обладать определенным доступом к внутренней кухне своего базового класса (посредством защищенных членов). Существуют, однако, и другие ситуации, когда взаимоотношения между объектами играют важную роль.

В этом разделе вкратце рассматриваются:

- **Содержание** (containment) — взаимоотношения, при которых один класс содержит в себе другой класс. Это очень напоминает наследование, но класс, содержащий в себе другой класс, может осуществлять контроль за доступом к членам содержащегося в нем класса и даже выполнять некоторую дополнительную обработку перед использованием членов вложенного класса.
- **Семейства** (collections) — когда некоторый класс выступает в качестве контейнера для множества экземпляров другого класса. Это напоминает массивы объектов, однако в данном случае имеются дополнительные возможности, включая индексирование, сортировку, изменение размеров и т. п.

Содержание

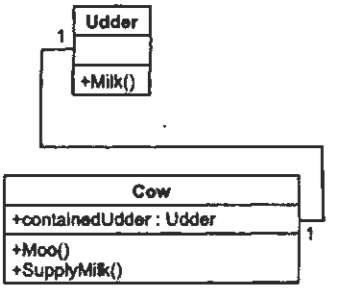
Применить содержание очень просто: достаточно использовать поле члена для хранения экземпляра объекта. Поле члена может быть общим; в этом случае пользователи объекта-контейнера смогут получить доступ к предоставляемым им методам и свойствам таким же образом, как и в случае наследования. Однако доступ к внутренним структурам класса посредством производного класса, имеющийся в случае наследования, отсутствует.

Существует альтернативная возможность объявить содержащийся объект как частный. В этом случае ни к одному из его членов не будет непосредственного доступа пользователей, даже если эти члены описаны как общие. Вместо этого мы

можем обеспечить доступ к этим членам посредством членов содержащего класса. Это означает, что мы получаем полный контроль над предоставлением членов содержащегося класса, если таковые существуют, а также можем производить дополнительную обработку в членах содержащего класса перед тем, как обеспечить доступ к членам содержащегося класса.

К примеру, класс `Cow` может содержать в себе класс `Udder` (вымя) с общим методом `Milk` (молоко). Объект `Cow` может обращаться к этому методу так, как ему это необходимо, например, как к составной части своего метода `SupplyMilk()`, однако эти детали не будут видимы (важны) для пользователей объекта `Cow`.

Содержащийся класс может быть изображен в UML с использованием ассоциативной линии. В случае простого содержания мы помечаем концы этой линии единицами, которые свидетельствуют о взаимоотношениях "один к одному" (один экземпляр объекта `Cow` содержит в себе один экземпляр объекта `Udder`). Мы можем также показать, что содержащийся объект класса `Udder` является частным полем класса `Cow` (см. рис. справа).



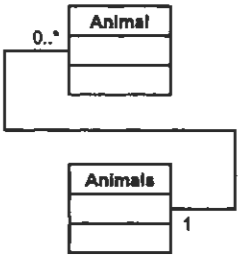
Семейства

Если вы помните, в главе 5 мы познакомились с использованием массивов для хранения нескольких переменных одного типа. Этот способ работает и в случае объектов (не забывайте, что те типы переменных, которые нами использовались, на самом деле являются объектами, поэтому в этом нет ничего удивительного). Например:

```
Animal[] animals = new Animal[5];
```

Семейства представляют собой массивы со всякими дополнениями. Семейства реализуются в виде классов во многом схоже с другими объектами. В качестве имени для них очень часто используется имя хранящихся в них объектов в форме множественного числа: например, класс с именем `Animals`, предназначенный для хранения семейства объектов `Animal`.

Основное отличие семейств от массивов заключается в том, что в семействах реализуются дополнительные функциональные возможности, такие как методы `Add()` (добавление) и `Remove()` (удаление), предназначенные для добавления и удаления элементов семейства. Обычно реализуется и свойство `Item` (элемент), которое возвращает элемент по его индексу. В большинстве случаев это свойство воплощается таким способом, который предусматривает и более изощренные методы доступа. Например, существует возможность организовать `Animals` так, чтобы к объектам можно было обращаться по имени.



В UML все это изображается так, как показано на рисунке слева. Здесь не представлены элементы, поскольку рисунок иллюстрирует взаимоотношения. Цифры на концах соединительной линии в данном случае означают, что в одном объекте `Animals` может содержаться ноль и более объектов `Animal`.

Более подробно семейства будут рассматриваться в главе 11.

Перегрузка операторов

В начале этой книги рассказывалось, каким образом операторы могут использоваться для выполнения манипуляций над переменными простых типов. Существуют ситуации, когда оказывается вполне логичным использовать эти операторы

для объектов, являющихся экземплярами наших собственных классов. Это вполне допустимо, поскольку в классах могут содержаться инструкции, касающиеся интерпретации этих операторов.

Например, мы можем добавить в наш класс `Animal` новое свойство с именем `Weight` (вес). В этом случае мы получаем возможность сравнивать вес животных:

```
if (cowA.Weight > cowB.Weight)
{
   . . .
}
```

Применив перегрузку оператора, мы можем включить в него логику, которая позволит использовать свойство `Weight` неявно, так что код можно будет записать следующим образом:

```
if (cowA > cowB)
{
   . . .
}
```

В этом случае оператор > является **перегружаемым**. Перегружаемый оператор — это такой оператор, для которого написан определяющий его работу код, являющийся составной частью определения одного из классов, в операциях над которыми он используется. В примере, приведенном выше, использованы два объекта класса `Cow`, поэтому определение перегружаемого оператора содержится в классе `Cow`. Мы можем аналогичным образом описать перегружаемые операторы для работы с объектами разных классов; в этом случае один из классов (или сразу оба) будет содержать соответствующий код.

Обратите внимание, что таким способом можно делать перегружаемыми только существующие в C# операторы, но не создавать новые. Однако мы можем создавать реализации как для унарного, так и для бинарного вариантов использования таких операторов типа оператора +.

С тем, каким образом этого можно добиться, мы познакомимся в главе 11.

События

Объекты могут создавать (и получать) **события** в процессе своей работы. События — это важные явления, с которыми мы можем работать в других местах кода и которые аналогичны исключительным ситуациям (но обладают более широкими возможностями). К примеру, нам может потребоваться, чтобы какой-либо специальный код выполнялся в тех случаях, когда объект `Animal` добавляется в семейство `Animals`, причем этот код не является ни составной частью класса `Animals`, ни кодом, в котором осуществляется вызов метода `Add()`. В этом случае нам придется воспользоваться **обработчиком событий**, который представляет собой специальный вид функции, вызывающейся в тот момент, когда происходит событие. При этом потребуется настроить этот обработчик таким образом, чтобы он ожидал наступления интересующих нас событий.

Используя события, мы можем создавать **управляемые ими приложения**, которые оказываются намного более продуктивными, чем вы можете сейчас представить. В качестве примера можно упомянуть, что большинство приложений Windows целиком построено на событиях. Любое нажатие кнопки или перемещение движка полосы прокрутки достигается путем обработки событий, наступление которых вызывается мышью или с клавиатуры.

В конце главы вы узнаете, каким образом события работают в приложениях Windows, а более детальное рассмотрение событий вас ждет в главе 12.

Сравнение ссылочных и значимых типов

Все данные в C# хранятся в переменных двумя способами, которые определяются типом переменной. Этот тип может относиться к одной из двух категорий: либо **ссылочный** тип, либо **значимый**. Вот в чем состоит различие между ними:

- **Значимые типы** хранятся сами и хранят свое содержимое в одном месте памяти (называемом стеком).
- **Ссылочные типы** содержат ссылку на какую-либо другую область памяти (она называется динамической), в которой хранится содержимое такой переменной.

На самом деле при использовании C# это не должно нас особенно беспокоить. До настоящего момента мы использовали переменные типа `string` (которые относятся к ссылочному типу) и другие простые переменные (большинство из которых относится к значимому типу, например, `int`) практически одинаково.

Среди простых типов ссылочными являются только `string` и `object`, хотя массивы также неявно представляют собой ссылочный тип. Каждый создаваемый нами класс будет относиться к ссылочному типу — именно поэтому мы и остановились на этом моменте.

Структуры

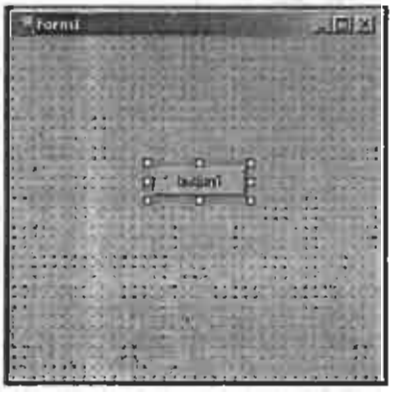
На данном этапе необходимо обратить внимание на один важный момент. Ключевым отличием структурных типов от классов является то, что первые представляют собой значимые типы.

Тот факт, что структурные типы и классы похожи друг на друга, возможно, стал вам ясен давно, особенно после того, как в главе 6 было сказано, что в структурных типах могут использоваться функции. Этот вопрос более подробно будет рассматриваться в следующей главе.

ООП в приложениях Windows

Возвращаясь к главе 2, мы можем вспомнить, каким образом в C# создаются простые приложения Windows. Они очень широко используют методы ООП, и в данном разделе мы рассмотрим эту тему, чтобы проиллюстрировать некоторые из наших утверждений. Для этого воспользуемся простым примером.

Практикум: объекты в действии



1. Создайте новое приложение Windows в директории `C:\BegCSharp\Chapter8` с именем `Ch08Ex01`.
2. Добавьте в приложение новое средство управления `Button` с помощью панели Toolbox и разместите его в центре Form1, как показано на рисунке слева.
3. Дважды щелкните по кнопке мышью для того, чтобы добавить код, предназначенный для обработки нажатия кнопки мыши. Внесите изменения в появившийся код, как показано ниже:

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

        private void button1_Click(object sender, System.EventArgs e)
        {
            ((Button)sender).Text = "Clicked!";
            Button newButton = new Button();
            newButton.Text = "New Button!";
            newButton.Click += new EventHandler(newButton_Click);
            Controls.Add(newButton);
        }

        private void newButton_Click(object sender, System.EventArgs e)
        {
            ((Button)sender).Text = "Clicked!!";
        }
    }
}
```

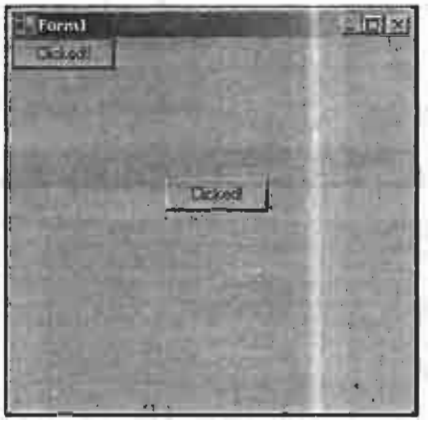
4. Запустите приложение (см. рис. справа).





5. Щелкните мышью по кнопке с надписью button1 (см. рис. слева).

6. Щелкните мышью по кнопке с надписью New Button! (см. рис. справа).



Как это работает

Написав всего несколько строк кода, мы создали Windows-приложение, которое выполняет некоторые действия, одновременно иллюстрируя использование определенных методов ООП в C#. Фраза "объектом может быть все, что угодно" становится еще более справедливой, когда дело касается Windows-приложений. Начиная от формы, в которой это выполняется, и заканчивая средствами управления в этой форме, нам постоянно приходится пользоваться методами ООП. В процессе описания этого примера автор выделил некоторые понятия; это сделано для того, чтобы уяснить их соотношение друг с другом.

Первое, что мы сделали в нашем приложении,— добавили в форму Form1 новую кнопку. Она представляет собой объект, называемый Button. Далее, щелкнув два раза мышью, мы добавили **обработчик событий**, который ожидает наступления события Click (нажатие), генерируемого объектом Button. Этот обработчик событий добавляется в код объекта Form, где инкапсулировано приложение, в качестве **частного метода**:

```
private void button1_Click(object sender, System.EventArgs e)
{
}
```

В качестве описателя используется ключевое слово private. Пока этих знаний вам должно быть достаточно; коды на C#, которые требуются для реализации методов ООП из настоящей главы, будут рассматриваться в главе 9.

Первая строка кода из числа добавленных нами изменяет текст на нажимаемой кнопке. В этом случае используется **полиморфизм**, с которым мы уже сталкивались. Объект Button, представляющий нажатую нами кнопку, передается обработчику событий как параметр типа object, тип которого мы изменяем на тип Button (это оказывается возможным, поскольку объект Button **наследуется** от класса .NET System.Object, для которого object является синонимом). Далее мы изменяем **свойство** объекта Text, чтобы изменить выводящийся текст:

```
((Button)sender).Text = "Clicked!";
```

Затем мы создаем новый объект Button с помощью ключевого слова new (обратите внимание, что в этом примере пространства имен заданы так, что позволяют использовать простой синтаксис, в противном случае нам бы пришлось вводить полностью квалифицированное имя объекта — System.Windows.Forms.Button):

```
Button newButton = new Button();
newButton.Text = "New Button!";
```

Затем в произвольное место программы добавляется **обработчик событий**, который будет использоваться для того, чтобы отреагировать на событие Click, генерируемое новой кнопкой:

```
private void newButton_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!!";
}
```

Затем мы регистрируем данный обработчик событий как ожидающий наступления события Click с помощью некоторого синтаксиса **перегрузки оператора**. Одновременно мы создаем новый объект EventHandler, используя для этого **конструктор не по умолчанию**, с именем новой функции обработчика событий:

```
newButton.Click += new EventHandler(newButton_Click);
```

В заключение мы используем объект **из семейства** Controls данной формы для того, чтобы добавить в форму новую кнопку с помощью метода Add():

```
Controls.Add(newButton);
```

Этот простой пример включает в себя почти все методы, рассмотренные в данной главе. Как вы можете заметить, в ООП нет ничего сложного, просто оно требует несколько иного угла зрения.

Итоги

В настоящей главе вы познакомились с полным описанием объектно-ориентированных методов. Мы изучали их в контексте программирования на C#, но это делалось в основном с иллюстративными целями. Подавляющее большинство понятий относится к ООП на любом языке программирования.

Мы начали с рассмотрения основ: например, какой смысл вкладывается в термин **"объект"** и каким образом объект может являться **экземпляром** класса. Затем вы познакомились с тем, что объекты могут иметь различные **члены**, такие как **поля**, **свойства** и **методы**. Доступ к этим членам может быть некоторым образом ограничен; в данном случае анализировалось, какой смысл вкладывается в понятия **общих** и **частных** членов. Далее говорилось о том, что члены могут быть **защищенными**, а также что их можно отнести к одной из двух категорий — **виртуальной** или **абстрактной** (причем абстрактные методы допустимы лишь для абстрактных классов). Также были рассмотрены различия между **статическими** (общего доступа) членами и членами **экземпляра** объекта.

После этого мы уделили некоторое внимание жизненному циклу объекта, включая использование **конструкторов** для создания объекта и **деструкторов** для его уничтожения. Далее, познакомившись с объединением членов в группы с целью создания интерфейсов, мы рассмотрели более совершенную процедуру уничтожения объектов — с помощью **удаляемых** объектов, поддерживающих интерфейс `IDisposable`.

Остальная часть главы была посвящена преимущественно рассмотрению возможностей ООП, многие из которых будут более подробно изучаться в последующих главах. Вы познакомились с **наследованием**, при котором классы **наследуются** от **базовых** классов, а также с двумя видами **полиморфизма** — через базовые классы и через интерфейсы; кроме того, вы узнали, каким образом можно использовать объекты для хранения одного или нескольких других объектов (посредством **содержания** или **семейств**). Наконец, было рассказано, как применять **перегрузку операторов** для упрощения синтаксиса при использовании объектов, а также каким образом объекты могут генерировать **события**.

Наконец, большая часть теории, изложенной в начале главы, была проиллюстрирована с помощью Windows-приложения.

В следующей главе мы познакомимся с тем, как определять классы в C#.

Упражнения

1. Какие из следующих слов являются ключевыми при описании уровней доступности в ООП?
   - ❑ Friend
   - ❑ Public
   - ❑ Secure
   - ❑ Private
   - ❑ Protected
   - ❑ Loose
   - ❑ Wildcard

2. Справедливо или нет следующее утверждение: "Мы должны вызывать деструкторы объектов вручную, иначе это приведет к ненужным тратам памяти".

3. Нужно ли создавать объект для того, чтобы использовать статический метод его класса?

4. Нарисуйте UML-диаграммы, аналогичные приведенным в настоящей главе, для следующих классов и интерфейса:

   - Абстрактный класс с именем `HotDrink`, у которого имеются методы `Drink()` (выпить), `AddMilk()` (добавить молоко) и `AddSugar()`, а также свойства `Milk` и `Sugar`.
   - Интерфейс с именем `ICup`, у которого имеются методы `Refill()` (наполнить повторно) и `Wash()` (вымыть), а также свойства `Color` (цвет) и `Volume` (емкость).
   - Класс с именем `CupOfCoffee`, который является производным от класса `HotDrink`, поддерживает интерфейс `ICup` и обладает дополнительным свойством `BeanType`.
   - Класс с именем `CupOfTea`, который является производным от класса `HotDrink`, поддерживает интерфейс `ICup` и обладает дополнительным свойством `LeafType`.

5. Напишите программу для функции, которая будет получать в качестве параметра объект одного из классов `CupOfCoffee` или `CupOfTea`. Далее функция должна вызывать методы `AddMilk()`, `Drink()` и `Wash()` для любого объекта, переданного ей в качестве параметра.



Определение классов

В предыдущей главе вы познакомились с возможностями объектно-ориентированного программирования (ООП). В этой главе мы перейдем от теории к практике и рассмотрим определение классов в C#.

Для начала мы не будем заходить столь далеко, чтобы определять члены классов, а сконцентрируем свое внимание на определении самих классов. Может показаться, что мы чересчур ограничиваем тему, но не волнуйтесь: в этой главе содержится достаточное количество материала, который вам необходимо усвоить.

В первую очередь мы рассмотрим основной синтаксис, применяемый при описании классов, ключевые слова, которые могут использоваться для определения режима доступа к классу и для других целей, а также способы задания наследования. Мы также разберем определение интерфейсов, поскольку оно во многих аспектах сходно с описанием классов.

Остальная часть главы будет посвящена изучению различных тем, могущих оказаться полезными при определении классов в C#, среди которых:

- Класс `System.Object`
- Полезные инструменты, предоставляемые Visual Studio.NET (VS)
- Библиотеки классов
- Сравнение интерфейсов и абстрактных классов
- Структурные типы
- Копирование объектов

Итак, мы начнем с того, что подробно ознакомимся с основами.

Определение классов в C#

В языке C# для описания классов используется ключевое слово `class`. При этом необходимо задействовать следующую основную конструкцию:

```
class myClass
{
   // члены класса
}
```

Этот код определяет класс с именем `myClass`. После того как мы определили класс, мы вольны создавать экземпляры этого класса в нашем проекте везде, где имеется доступ к этому определению. По умолчанию классы определяются как

internal (внутренние), что означает, что доступ к ним будет иметь только код текущего проекта. Мы можем указать это явно, воспользовавшись ключевым словом определения доступа `internal` (хотя и не обязаны этого делать):

```
internal class MyClass
{
   // члены класса
}
```

При желании можно указать, что класс является общим и должен быть доступен из других проектов. Для этого используется ключевое слово `public`:

```
public class MyClass
{
   // члены класса
}
```

Обратите внимание, что классы, которые объявляются самостоятельно, не могут быть частными или защищенными. Соответствующие модификаторы — private и protected — можно использовать только для описания классов, являющихся членами других классов. Их мы рассмотрим в следующей главе.

Кроме ключевых слов этих двух модификаторов доступа, для описания класса можно использовать ключевое слово `abstract` (**абстрактный**; создавать экземпляры такого класса запрещено, он может только наследоваться, в нем допускается наличие абстрактных членов) или `sealed` (**изолированный**; такой класс не может наследоваться). Эти ключевые слова являются взаимоисключающими. Таким образом, абстрактный класс должен определяться следующим образом:

```
public abstract class MyClass
{
   // члены класса, среди которых могут быть абстрактные
}
```

В данном случае `MyClass` является общим абстрактным классом, хотя также возможны и внутренние абстрактные классы.

Изолированные классы определяются следующим образом:

```
public sealed class MyClass
{
   // члены класса
}
```

Так же, как абстрактные классы, изолированные классы могут быть общими или внутренними.

В определении класса может также задаваться и наследование. Для этого следует после имени класса поместить двоеточие, за которым должно следовать имя базового класса. Например:

```
public class MyClass : MyBase
{
   // члены класса
}
```

Заметьте, что в определении класса в C# допускается наличие *только одного* базового класса, и если данный класс наследуется от некоторого абстрактного класса, то в нем *должны* быть реализованы все наследуемые абстрактные члены (если, конечно, этот класс сам не является абстрактным).

Компилятор не допустит ситуации, при которой производный класс будет более доступным, чем его базовый класс. Отсюда следует, что внутренний класс *может* наследоваться от общего базового класса, но общий класс *не может* наследоваться от внутреннего базового класса. Следующий код является допустимым:

```
public class MyBase
{
   // члены класса
}

internal class MyClass : MyBase
{
   // члены класса
}
```

А такой код не пройдет компиляцию:

```
internal class MyBase
{
   // члены класса
}

public class MyClass : MyBase
{
   // члены класса
}
```

Если базовый класс не задан, то класс наследуется только от базового класса `System.Object` (для которого в C# используется синоним `object`). `System.Object` безусловно является корневым в иерархии наследования для *всех* классов. Мы будем изучать этот фундаментальный класс ниже.

Помимо указания базовых классов, мы можем таким же способом — через двоеточие — указать и поддерживаемые интерфейсы. Если одновременно требуется задавать базовый класс, то он должен располагаться первым после двоеточия, а интерфейсы — за ним. Если базовый класс не задается, то интерфейсы указываются непосредственно за двоеточием. Для отделения имени базового класса (если таковое присутствует) и для отделения одних имен интерфейсов от других следует использовать запятую.

Например, мы можем добавить в класс `MyClass` интерфейс:

```
public class MyClass : IMyInterface
{
   // члены класса
}
```

Все члены, являющиеся интерфейсами, *должны* иметь реализацию в любом классе, поддерживающем данный интерфейс, хотя, конечно, всегда существует возможность реализовать "пустой" интерфейс (без какого бы то ни было функционального кода), если от данного члена не требуется выполнения каких-либо действий.

Следующее объявление неверно, поскольку базовый класс `MyBase` не является первым вхождением списка наследования:

```
public class MyClass : IMyInterface, MyBase
{
   // члены класса
}
```

Правильный способ задать и базовый класс, и интерфейс следующий:

```
public class MyClass : MyBase, IMyInterface
{
   // члены класса
}
```

Не забывайте, что допускается наличие нескольких интерфейсов, поэтому следующий код также является допустимым:

```
public class MyClass : MyBase, IMyInterface, IMySecondInterface
{
   // члены класса
}
```

Для того чтобы вы смогли быстро освежить свою память, ниже приводится таблица допустимых в определениях классов комбинаций модификаторов доступа:

Модификатор	Значение
Отсутствует либо `internal`	Класс доступен только в рамках текущего проекта
`public`	Класс доступен отовсюду
`abstract` или `internal abstract`	Класс доступен только в рамках текущего проекта, не допускает создания экземпляров, может только наследоваться
`public abstract`	Класс доступен отовсюду, не допускает создания экземпляров, может только наследоваться
`sealed` или `internal sealed`	Класс доступен только в рамках текущего проекта, не может наследоваться, допускает только создание экземпляров
`public sealed`	Класс доступен отовсюду, не может наследоваться, допускает только создание экземпляров

Определение интерфейсов

Интерфейсы объявляются тем же способом, что и классы, только вместо ключевого слова `class` используется ключевое слово `interface`. Например:

```
interface IMyInterface
{
   // члены интерфейса
}
```

Ключевые слова для модификации доступа `public` и `internal` используются точно так же, поэтому для того, чтобы сделать интерфейс общедоступным, следует использовать ключевое слово `public`:

```
public interface IMyInterface
{
   // члены интерфейса
}
```

Для интерфейсов ключевые слова `abstract` и `sealed` использовать нельзя, так как ни один модификатор для интерфейсов не имеет смысла (у них отсутствует реализация, следовательно, для них не могут создаваться экземпляры в явном виде).

Наследование для интерфейсов определяется аналогично наследованию для классов. Основное отличие здесь в том, что мы можем использовать интерфейсы с множественными базами, например:

```
public interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
{
   // члены интерфейса
}
```

Интерфейсы, как и классы, наследуются от `System.Object`. Этот механизм допускает полиморфизм интерфейсов. Однако, как отмечалось ранее, нельзя создать экземпляр интерфейса таким же способом, как и экземпляр класса.

Давайте обратимся к некоторым примерам определения классов, которые приводятся вместе с кодом, в котором они используются.

Практикум: определение классов

1. Создайте новое консольное приложение с именем `Ch09Ex01` в директории `C:\BegCSharp\Chapter9`.

2. Измените код в `Class1.cs` следующим образом:

```
namespace Ch09Ex01
{
   public abstract class MyBase
   {
   }

   internal class MyClass : MyBase
   {
   }

   public interface IMyBaseInterface
   {
   }

   internal interface IMyBaseInterface2
   {
   }

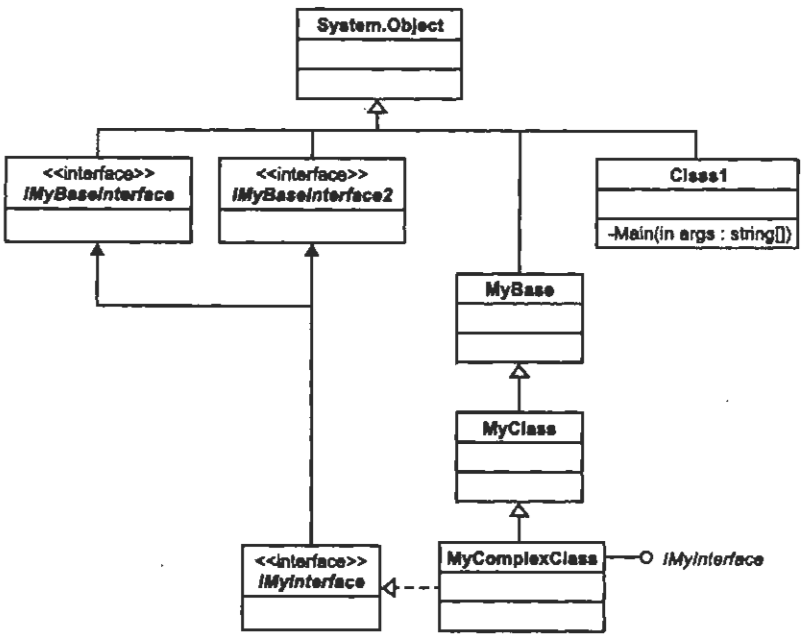
   internal interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
   {
   }

   internal sealed class MyComplexClass : MyClass, IMyInterface
   {
   }

   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      static void Main(string[] args)
      {
         MyComplexClass myObj = new MyComplexClass();
         Console.WriteLine(myObj.ToString());
      }
   }
}
```

3. Выполните проект:





Как это работает

В данном проекте описаны классы и интерфейсы, обладающие иерархией наследования, показанной на рисунке слева.

Мы включили сюда `Class1`, так как его описание аналогично описанию остальных наших классов, хотя он и не входит в основную иерархию классов. Метод `Main()`, находящийся в этом классе, является точкой входа в приложение; этот момент обсуждался раньше.

`MyBase` и `IMyBaseInterface` определены как общие, поэтому они доступны из других проектов. Остальные классы и интерфейсы являются внутренними и доступны только в рамках настоящего проекта.

В программе вызывается метод `ToString()` объекта `myObj`, который является экземпляром класса `MyComplexClass`:

```
MyComplexClass myObj = new MyComplexClass();
Console.WriteLine(myObj.ToString());
```

Это один из наследуемых от `System.Object` методов (класс `System.Object` не изображен на диаграмме просто для краткости). `ToString()` возвращает имя класса данного объекта в виде строки.

System.Object

Поскольку все классы наследуются от `System.Object`, то они обладают доступом ко всем его защищенным и общим членам. Поэтому имеет смысл ознакомиться с тем, что нам доступно. `System.Object` содержит следующие методы:

Метод	Возвращаемый тип	Виртуальный	Статический	Описание
`Object()`	Отсутствует	Нет	Нет	Конструктор типа `System.Object`. Автоматически вызывается конструкторами производных типов
`~Object()` (также известен под именем `Finalize()` — см. следующий раздел)	Отсутствует	Нет	Нет	Деструктор для типа `System.Object`. Автоматически вызывается деструкторами производных типов, сам по себе вызван быть не может
`Equals(object)`	`bool`	Да	Нет	Сравнивает объект, для которого вызывается, с другим объектом и возвращает значение `true`, если они равны. Реализация, выполняющаяся по умолчанию, проверяет, *ссылается ли*

Продолжение таблицы

Метод	Возвращаемый тип	Виртуальный	Статический	Описание
`Equals(object)` (*продолжение*)				переданный в качестве параметра объект на тот же самый объект (поскольку объекты представляют собой ссылочные типы). Если необходимо сравнивать объекты каким-либо иным образом, например, на предмет одинакового значения, этот метод может быть переопределен
`Equals(object, object)`	`bool`	Нет	Да	Сравнивает два объекта, передаваемых ему в качестве параметров, на предмет того, равны ли они. Эта проверка выполняется с помощью метода `Equals(object)`. Заметьте, что, если оба объекта обладают нулевыми ссылками, возвращается значение `true`
`ReferenceEquals(object, object)`	`bool`	Нет	Да	Сравнивает два переданных ему объекта, определяя, являются ли они ссылками на один и тот же экземпляр
`ToString()`	`string`	Да	Нет	Возвращает строку, соответствующую экземпляру объекта. По умолчанию это квалифицированное имя класса (см. предыдущий пример), однако метод можно переопределить для выполнения действий, подходящих для типа данного класса
`MemberwiseClone()`	`object`	Нет	Нет	Копирует объект посредством создания нового экземпляра объекта и копирования всех членов. Обратите внимание, что такое копирование членов *не приводит* к созданию новых экземпляров этих членов. Любые члены ссылочного типа в новом объекте будут ссылаться на те же объекты, на которые они ссылаются в исходном классе. Рассматриваемый метод является защищенным, поэтому его можно использовать внутри класса или внутри производных классов
`GetType()`	`System.Type`	Нет	Нет	Возвращает тип объекта в виде объекта `System.Type`
`GetHashCode()`	`int`	Да	Нет	Используется как функция хеширования для объектов. **Хеш-функция** — это функция, возвращающая значение, которое позволяет идентифицировать объект в некоторой сжатой форме

Эти методы являются основными, они должны поддерживаться всеми типами объектов в .NET Framework, хотя, возможно, некоторые из них вам никогда не придется использовать (или только при определенных обстоятельствах, как, например, `GetHashCode()`).

Метод `GetType()` полезен при использовании полиморфизма, поскольку он позволяет выполнять разные коды для объектов разных типов, а не один и тот же код для всех объектов, как это часто бывает. Например, если у нас имеется функция,

которой передается параметр типа `object` (это означает, что мы можем передавать ей практически все, что угодно), то можно предусмотреть выполнение в ней специальных работ в случае поступления объектов конкретного типа. Воспользовавшись сочетанием `GetType()` с `typeof()` (оператор C#, который преобразовывает имя класса в объект `System.Type`), мы получаем возможность выполнять сравнения примерно следующим образом:

```
if (typeof(myObj) == typeof(MyComplexClass))
{
   // Объект myObj является экземпляром класса MyComplexClass
}
```

Возвращаемый объект `System.Type` обладает гораздо более широкими возможностями, но здесь мы не будем на нем останавливаться. Отражение подробно описывается в главе 22.

Очень часто оказывается весьма полезным переопределить метод `ToString()`, особенно в тех ситуациях, когда содержимое объекта может быть с легкостью представлено в виде одной удобочитаемой строки.

Эти методы `System.Object` будут постоянно встречаться в последующих главах, поэтому сейчас мы завершаем обсуждение; дополнительные детали будут объясняться по мере необходимости.

Конструкторы и деструкторы

Когда мы описываем какой-либо класс в C#, то в большинстве случаев нет необходимости описывать соответствующие ему конструкторы и деструкторы, поскольку объект базового класса `System.Object` обеспечивает их реализацию по умолчанию. Однако иногда стоит описать их самостоятельно, что позволит инициализировать и уничтожать объекты.

Для того чтобы добавить в класс простой конструктор, необходимо воспользоваться следующим несложным синтаксисом:

```
class MyClass
{
   public MyClass()
   {
      // Код конструктора
   }
}
```

Такой конструктор обладает тем же именем, что и класс, в котором он содержится, не имеет параметров (что превращает его в конструктор, использующийся по умолчанию) и является общим, что позволяет создавать экземпляры объектов данного класса (более подробная информация по этому вопросу содержится в предыдущей главе).

Существует также возможность использовать частный конструктор по умолчанию, что означает, что экземпляры объектов данного класса не могут создаваться с помощью этого конструктора (еще раз отсылаем читателей к предыдущей главе):

```
class MyClass
{
   private MyClass()
   {
      // Код конструктора
   }
}
```

Кроме того, у нас имеется возможность аналогичным образом определить конструкторы данного класса, использующиеся не по умолчанию; для этого достаточно просто задать параметры. Например:

```
class MyClass
{
   public MyClass()
   {
      // Код конструктора, используемого по умолчанию
   }

   public MyClass(int myInt)
   {
      // Код конструктора, используемого не по
      // умолчанию (используется параметр myInt)
   }
}
```

Колнчество задаваемых конструкторов не ограничено.

Деструкторы определяются с помощью несколько иного синтаксиса. Деструктор, нспользуемый в .NET (н предоставляемый классом `System.Object`), называется `Finalize()`, однако для объявлення деструктора используется другое имя. Вместо того чтобы переопределять `Finalize()`, мы используем следующий код:

```
class MyClass
{
   ~MyClass()
   {
      // тело деструктора
   }
}
```

Код, заключенный в деструкторе, будет выполняться при сборке мусора, позволяя освобождать удержнваемые ресурсы. После вызова деструктора происходят явные вызовы деструкторов базовых классов, включая вызов `Finalize()` в корневом классе `System.Object`. Такой способ позволяет .NET Framework гарантнровать выполненне, поскольку переопределенне `Finalize()` означало бы, что необходимы явные вызовы базовых классов, а это тант в себе потенциальную опасность (мы познакомнмся с вызовом методов базовых классов в следующей главе).

Последовательность выполнения конструкторов

Если нам требуется выполнить несколько действий в конструкторах класса, то удобнее поместнть весь этот код в одном месте, что дает те же преимущества, что и разбиение кода на функцни. Для этого можно использовать отдельный метод (см. следующую главу), одиако в C# имеется замечательная альтернативная возможность. Каждый конструктор может быть настроен таким образом, что перед выполнением собственного кода он будет вызывать некоторый другой конструктор.

Однако прежде чем перейти к рассмотрению этого вопроса, давайте поближе познакомимся с тем, что по умолчанию происходнт при создании нового экземпляра класса.

Для того чтобы создать экземпляр производного класса, необходимо создать экземпляр его базового класса. В свою очередь, чтобы создать экземпляр этого базового класса, требуется создать экземпляр базового класса этого базового класса — и так далее до `System.Object`. В результате, какой бы конструктор ни использовался для создания класса, `System.Object.Object()` всегда будет вызываться первым.

Если используется конструктор класса не по умолчанию, то в таком случае по умолчанию будет использоваться конструктор базового класса, сигнатура которого совпадает с сигнатурой данного конструктора. Если таковой обнаружить не удается, то используется конструктор базового класса по умолчанию (это происходит всегда, за исключением корневого класса `System.Object`, поскольку у него отсутствуют конструкторы, использующиеся не по умолчанию). Давайте рассмотрим некоторый пример, который поможет проиллюстрировать последовательность событий. 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)
   {
   }
}
```

Если мы попытаемся создать экземпляр класса `MyDerivedClass` следующим образом:

```
MyDerivedClass myObj = new MyDerivedClass();
```

то это приведет к такой последовательности событий:

- Выполнится конструктор `System.Object.Object()`.
- Выполнится конструктор `MyBaseClass.MyBaseClass()`.
- Выполнится конструктор `MyDerivedClass.MyDerivedClass()`.

Если же мы попытаемся создать экземпляр класса таким образом:

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

то соответствующая последовательность будет иметь следующий вид:

- Выполнится конструктор `System.Object.Object()`.
- Выполнится конструктор `MyBaseClass.MyBaseClass(int i)`.
- Выполнится конструктор `MyDerivedClass.MyDerivedClass(int i)`.

Наконец, если мы воспользуемся следующим вариантом:

```
MyDerivedClass myObj = new MyDerivedClass(4, 8);
```

то произойдет вот что:

- Выполнится конструктор `System.Object.Object()`.
- Выполнится конструктор `MyBaseClass.MyBaseClass()`.
- Выполнится конструктор `MyDerivedClass.MyDerivedClass(int i, int j)`.

При таком подходе мы получаем возможность поместить код, ответственный за обработку параметра `int i`, в `MyBaseClass(int i)`, подразумевая, что конструктору `MyDerivedClass(int i, int j)` достанется меньше работы — ему придется обрабатывать только параметр `int j`. (Подобные рассуждения строятся на основе предположения, что параметр `int i` в обоих случаях имеет один и тот же смысл, что может и не выполняться, хотя на практике при таком способе оформления обычно выполняется.) C# при необходимости позволяет нам задать именно такой тип поведения.

Для этого требуется просто указать конструктор базового класса в определении конструктора нашего производного класса следующим образом:

```
public class MyDerivedClass : MyBaseClass
{
   . . .

   public MyDerivedClass(int i, int j) : base(i)
   {
   }
}
```

Ключевое слово `base` указывает .NET, что в процессе создания экземпляра следует использовать конструктор базового класса с сигнатурой, совпадающей с заданной. В данном случае мы задействуем единственный параметр `int`, поэтому будет вызван `MyBaseClass(int i)`. Конструктор `MyBaseClass()` не будет вызываться, т. е. последовательность событий будет такой же, как и в последнем примере,— что, собственно, в данном случае и требовалось.

Существует возможность с помощью этого же ключевого слова задавать литеральные значения для конструкторов базового класса, допустим, применяя конструктор класса `MyDerivedClass`, использующийся по умолчанию, для вызова конструктора класса `MyBaseClass`, использующегося не по умолчанию:

```
public class MyDerivedClass : MyBaseClass
{
   public MyDerivedClass() : base(5)
   {
   }

   . . .
}
```

В этом случае последовательность событий будет иметь такой вид:

- Выполнится конструктор `System.Object.Object()`.
- Выполнится конструктор `MyBaseClass.MyBaseClass(int i)`.
- Выполнится конструктор `MyDerivedClass.MyDerivedClass()`.

Кроме ключевого слова `base`, в этом контексте может использоваться еще одно ключевое слово: `this`. Оно указывает процессу создания экземпляра в .NET на необходимость использовать конструктор не по умолчанию для *текущего* класса, прежде чем будет вызван указанный конструктор. Например:

```
public class MyDerivedClass : MyBaseClass
{
   public MyDerivedClass() : this(5, 6)
   {
   }

   . . .

   public MyDerivedClass(int i, int j) : base(i)
   {
   }
}
```

Это приведет к такой последовательности событий:

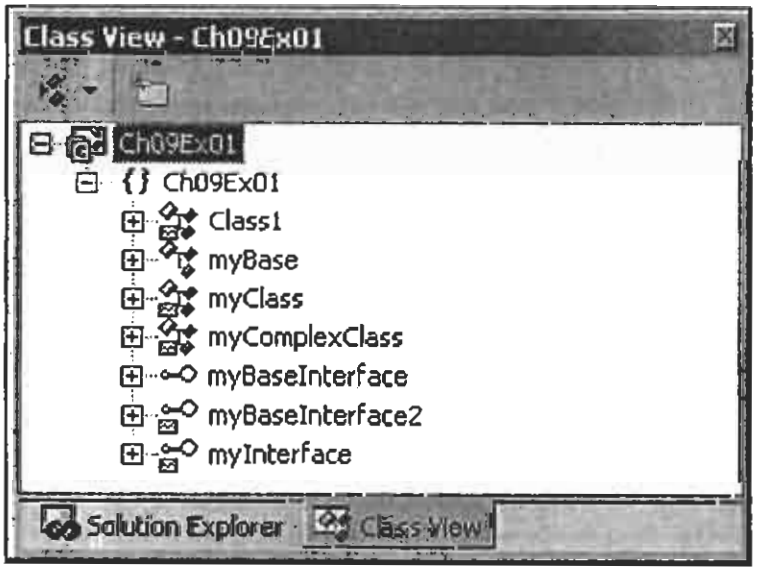
- Выполнится конструктор `System.Object.Object()`.
- Выполнится конструктор `MyBaseClass.MyBaseClass(int i)`.
- Выполнится конструктор `MyDerivedClass.MyDerivedClass(int i, int j)`.
- Выполнится конструктор `MyDerivedClass.MyDerivedClass()`.

Единственным ограничением в данном случае является задание только одного конструктора, использующего ключевые слова `base` или `this`. На самом деле, как было продемонстрировано в предыдущем примере, это не такое уж серьезное ограничение, поскольку все равно остается возможность конструирования чрезвычайно изощренных последовательностей выполняемых действий.

Мы увидим этот способ в действии немного позже.

Инструменты ООП в Visual Studio.NET

Поскольку ООП является основополагающим понятием .NET Framework, то в VS предусмотрены определенные инструменты, облегчающие разработку приложений с использованием ООП. В этом разделе будут рассмотрены некоторые из них.

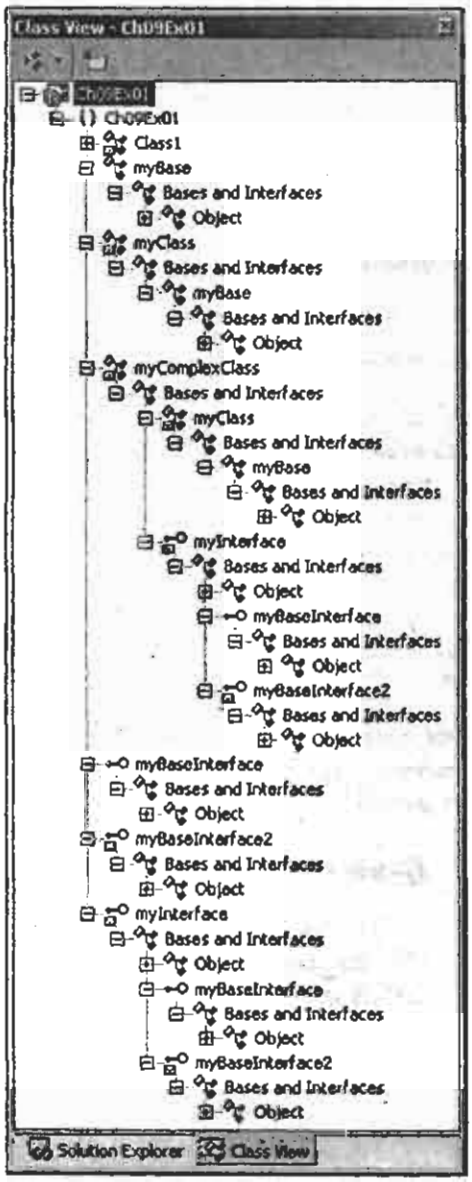


Окно Class View

В главе 2 мы видели, что окно Solution Explorer делит экранное пространство с окном Class View (окно просмотра классов, в котором отображается иерархия классов приложения. Кроме того, названное окно позволяет моментально узнать характеристики используемых классов. Существует несколько режимов просмотра этой информации, причем по умолчанию используется режим Sort By Type (упорядочивание по типу). Для проекта, который был взят в качестве примера в предшествующем разделе, это окно будет иметь вид, подобный представленному на рисунке слева.

В окне **Class View** может использоваться большое количество разнообразных символов, среди которых:

- — Проект
- — Пространство имен
- — Класс
- — Интерфейс
- — Метод
- — Свойство
- — Поле
- — Структура
- — Перечислимый тип
- — Элемент перечислимого типа
- — Событие



Обратите внимание, что часть символов используется для определения типов, не являющихся классами, например, для перечислений или структур.

Некоторые из вхождений могут иметь по дополнительному символу, расположенному ниже и обозначающему режим доступа к ним (у вхождений общего типа символ отсутствует):

- — Частный
- — Защищенный
- — Внутренний

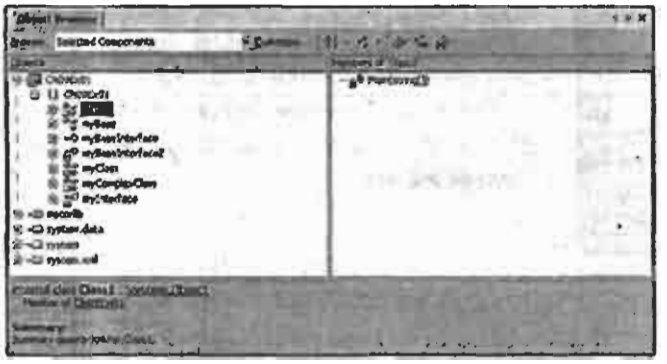
Для обозначения абстрактных, изолированных и виртуальных вхождений никаких символов не используется.

Все режимы работают одинаково, позволяя разворачивать определения классов с использованием стандартных средств управления просмотром дерева. Классы и интерфейсы разворачиваются вплоть до уровня `System.Object` (см. рис. слева). Отсюда можно почерпнуть любую информацию о классах проекта.

Помимо просмотра этой информации, в данном окне можно получить доступ к коду большинства элементов. Двойной щелчок мышью по элементу или двойное нажатие правой кнопки мыши и выбор пункта меню **Go To Definition** (переход к определению) позволяет перейти непосредственно к тому коду, который описывает данный элемент, если этот код является доступным. В ином случае — например, если он находится в каком-либо базовом типе, к которому нет доступа,— мы перейдем в окно **Object Browser** (браузер объектов).

Окно Object Browser

Окно Object Browser — это расширенная версия окна Class View, которая позволяет просматривать другие классы, доступные нашему проекту, и даже совершенно посторонние классы. Переход в это окно осуществляется либо автоматически (например, в ситуациях, подобных описанной в предшествующем разделе), либо вручную через View | Other Windows | Object Browser. Это окно появляется в основном окне и позволяет осуществлять просмотр так же, как и а окне Class View.



В окне Object Browser (см. рис. слева), в отличие от окна Class View, классы и члены классов изображаются в различных местах, при этом в их число включаются все модули .NET, на которые ссылается данный проект. Здесь можно просматривать, например, вхождения пространства имен `System`.

После того как некоторый элемент выбран, в поле внизу отображается информация о нем. В данном случае в ней содержится уровень доступа, базовый класс и пространство имен для класса `Class1`. Эту информацию также можно использовать для поиска: щелчок мышью, например, на `System.Object` приведет к выводу информации, относящейся к данному классу. Кроме того, здесь можно просмотреть некоторую обобщенную информацию. Она генерируется с использованием документирующих XML-комментариев, включенных в код:

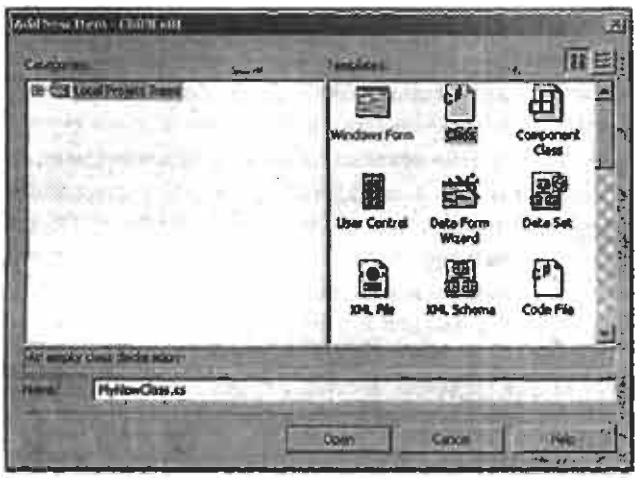
```
/// <summary>
/// Краткое описание класса Class1.
/// </summary>
class Class1
{
   static void Main(string[] args)
   {
      MyComplexClass myObj = new MyComplexClass();
      Console.WriteLine(myObj.ToString());
   }
}
```

Более подробно документирующие XML-комментарии будут рассматриваться в главе 18.

Добавление классов

В состав VS входят инструменты, которые позволяют ускорять выполнение некоторых наиболее распространенных задач, и некоторые из них применимы для ООП. Один из таких инструментов позволяет быстро добавлять в проект новые классы при минимуме информации, которую необходимо набирать.

Доступ к этому инструменту можно получить либо через пункт меню File | Add New Item..., либо с помощью двойного щелчка мышью на имени проекта в окне Solution Explorer и выбора подходящего элемента. В обоих вариантах появляется диалоговое окно, которое позволяет выбрать тип элемента, который требуется добавить. Для того чтобы добавить класс, необходимо выбрать вхождение Class в окне, расположенном справа, ввести имя файла, в котором будет содержаться



этот класс, и затем нажать **Open**. В результате создаваемому классу будет присвоено имя в соответствии с именем, выбранным для файла (см. рис. слева).

В примере, приводившемся в этой главе, мы добавляли описания классов в файл `Class1.cs` вручную. Однако часто хранение различных классов в разных файлах позволяет упростить контроль за ними.

Когда мы откроем проект `Ch09Ex01`, мы увидим, что в файле `MyNewClass.cs` появился следующий код:

```
using System;

namespace Ch09Ex01
{
    /// <summary>
    /// Краткое описание класса MyNewClass.
    /// </summary>
    public class MyNewClass
    {
        public MyNewClass()
        {
            //
            // Следует сделать: вставить сюда логику конструктора
            //
        }
    }
}
```

Данный класс — `MyNewClass` — описывается в том же пространстве имен, что и наш класс — `Class1`, являющийся точкой входа, поэтому мы можем использовать его из программы так же, как если бы он был определен в том же самом файле.

Как можно увидеть из кода (или, точнее, из находящихся в коде комментариев), созданный класс содержит конструктор по умолчанию.

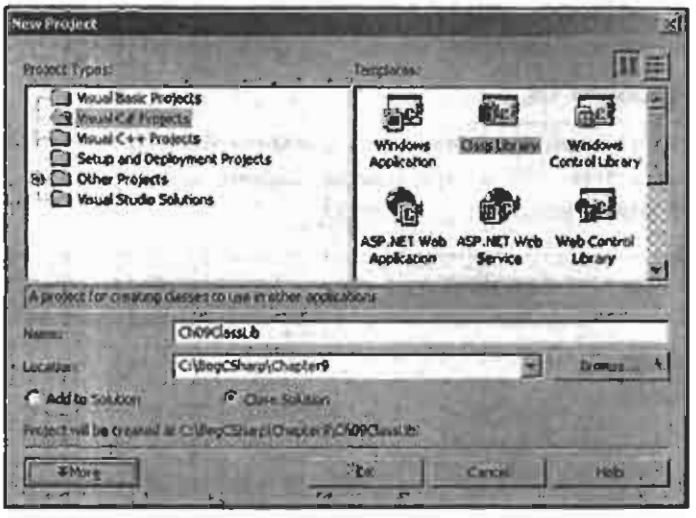
Проекты, представляющие собой библиотеки классов

Классы можно размещать не только в отдельных файлах в рамках одного проекта, но и в совершенно независимых проектах. Проект, в котором не содержится ничего, кроме классов (вместе с описаниями других относящихся к делу типов, но без точки входа), называется **библиотекой классов**.

Проекты, представляющие собой библиотеки классов, компилируются в модули `.dll`, а доступ к их содержимому можно получить, включая в другие проекты ссылки на них (которые могут быть, хотя и не обязательно, частью того же самого решения). Такой подход позволяет увеличить степень инкапсуляции, обеспечиваемой объектами, поскольку процесс изменения и совершенствования библиотек классов может происходить, абсолютно не затрагивая те проекты, которые используют данные библиотеки. Это позволяет с легкостью совершенствовать услуги, предоставляемые различными классами (что может влиять сразу на несколько различных приложений, использующих эти классы).

Давайте рассмотрим пример проектов, один из которых представляет собой библиотеку классов, а другой использует классы, которые содержатся в этой библиотеке.

Практикум: использование библиотеки классов



1. Создайте новый проект с именем `Ch09ClassLib` в директории `C:\BegCSharp\Chapter9`, воспользовавшись диалоговым окном **New Project** (см. рис. слева).
2. Переименуйте файл `Class1.cs` в `MyExternalClass.cs` (это можно сделать, щелкнув правой кнопкой мыши по имени файла в окне **Solution Explorer** и выбрав опцию **Rename** (переименовать)).
3. Измените код в `MyExternalClass.cs` так, чтобы он отражал данное изменение имени класса. Для этого запишем код так:

```
public class MyExternalClass
{
   public MyExternalClass()
   {
      //
      // Следует сделать: вставить сюда логику конструктора
      //
   }
}
```

4. Добавьте в проект новый класс, использовав в качестве имени файла `MyInternalClass.cs`.
5. Модифицируйте код таким образом, чтобы класс `MyInternalClass` стал внутренним:

```
internal class MyInternalClass
{
   public MyInternalClass()
   {
      //
      // Следует сделать: вставить сюда логику конструктора
      //
   }
}
```

6. Откомпилируйте проект (обратите внимание, что в этом проекте не имеется точки входа, поэтому его невозможно запустить как обычный проект — вместо этого вы можете просто построить его, воспользовавшись опцией меню **Build | Build**).
7. Создайте новое консольное приложение с именем `Ch09Ex02` в директории `C:\BegCSharp\Chapter9`.



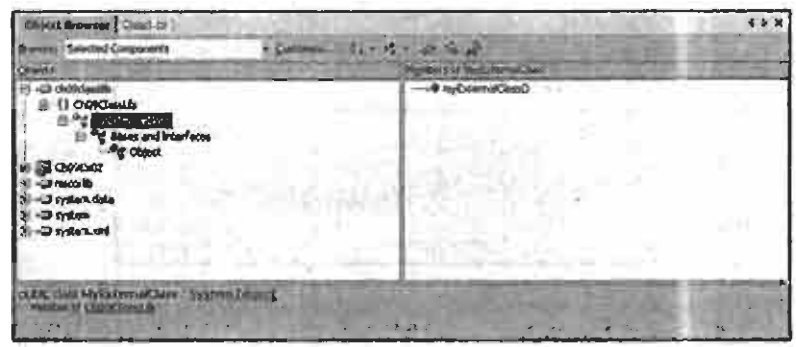
8. Выберите пункт меню Project | Add Reference... либо выберите ту же самую опцию, дважды щелкнув правой кнопкой мыши по пункту References в окне Solution Explorer.

9. Щелкните по кнопке Browse..., перейдите в директорию `C:\BegCSharp\Chapter9\Ch09ClassLib\bin\Debug\` и дважды щелкните мышью по `Ch09ClassLib.dll`.

10. Нажмите OK.

11. После того как выполнение операции будет завершено, убедитесь, что в окне Solution Explorer добавилась новая ссылка (см. рис. слева).

12. Откройте окно Object Browser и посмотрите, какие объекты содержатся в новой ссылке (см. рис. справа).



13. Внесите следующие изменения в `Class1.cs`:

```
using System;
using Ch09ClassLib;

namespace Ch09Ex02
{
    /// <summary>
    /// Краткое описание класса Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            MyExternalClass myObj = new MyExternalClass();
            Console.WriteLine(myObj.ToString());
        }
    }
}
```

14. Запустите прнложение:



Как это работает

В этом практикуме мы создали два проекта, один из которых является библиотекой классов, а второй — консольным приложением. Проект, представляющий собой библиотеку классов,— `Ch09ClassLib` — включает два класса: `MyExternalClass`,

являющийся общедоступным, и `MyInternalClass`, к которому имеется только внутренний доступ. Консольное приложение — `Ch09Ex02` — содержит простой код, использующий библиотеку классов.

Для того чтобы использовать классы из `Ch09ClassLib`, мы добавили в приложение `Ch09Ex02` ссылку на DLL, которая была создана проектом библиотеки классов. Для этого мы просто указали соответствующую DLL. Для задач, решаемых в данном примере, оказалось достаточно указать на выходной файл библиотеки классов, однако с тем же успехом мы могли бы скопировать этот файл в такое место, где он был бы локальным по отношению к `Ch09Ex02`. Это позволило бы продолжать разработку библиотеки классов, не затрагивая консольное приложение. Для того чтобы заменить старую версию модуля на новую, потребуется записать вновь созданную DLL на место старой.

После добавления ссылки мы получили возможность просмотреть перечень доступных классов с помощью Object Browser. Поскольку один из двух классов — `MyInternalClass` — является внутренним, то он не виден: этот класс недоступен внешним проектам. Напротив, класс `MyExternalClass` доступен, и мы используем его в консольном приложении.

Мы можем попробовать заменить код в консольном приложении на код, который попытается использовать внутренний класс следующим образом:

```
static void Main(string[] args)
{
   MyInternalClass myObj = new MyInternalClass();
   Console.WriteLine(myObj.ToString());
}
```

При попытке откомпилировать такой код будет получено следующее сообщение об ошибке:

```
C:\BegCSharp\Chapter9\Ch09Ex02\Class1.cs(13): 'Ch09ClassLib.MyInternalClass'
(C:\BegCSharp\Chapter9\Ch09Ex02\Class1.cs(13): класс 'Ch09ClassLib.MyInternalClass'
is inaccessible due to its protection level
недоступен из-за уровня его защиты)
```

Этот способ использования классов из внешних модулей является ключевым при программировании на C# в .NET Framework. Фактически так мы поступаем со всеми классами в .NET Framework, поскольку работа с ними организована аналогичным образом.

Сравнение интерфейсов и абстрактных классов

В этой главе мы познакомились с тем, каким образом можно создавать и интерфейсы, и абстрактные классы (пока не содержащие членов — к рассмотрению этого вопроса мы перейдем в следующей главе). Эти два типа в некотором смысле аналогичны, и имеет смысл рассмотреть те ситуации, в которых может потребоваться использование одних или других. Итак, сначала о сходстве.

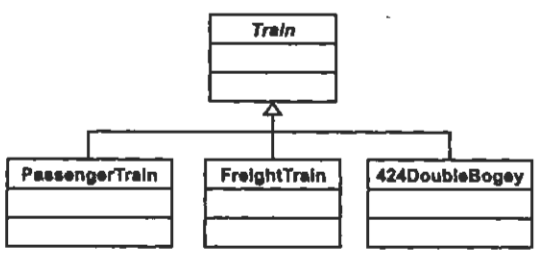
Как в абстрактных классах, так и в интерфейсах допускается наличие членов, которые могут наследоваться производными классами. Ни абстрактные классы, ни интерфейсы не допускают непосредственного создания экземпляров, хотя мы можем объявлять переменные соответствующих типов. В этом случае можно использовать полиморфизм при присваивании объектов, которые наследуются от этих типов, переменным таких типов. В обоих случаях мы получаем возможность использовать члены этих типов с помощью этих переменных, хотя и не имеем непосредственного доступа к другим членам производного класса.

Теперь о различиях.

Производные классы могут наследоваться только от одного базового класса, что означает, что непосредственно наследоваться может только один абстрактный класс (хотя вся цепочка наследования может включать в себя несколько абстрактных классов). Напротив, классы могут использовать произвольное количество интерфейсов. Отличие, однако, не столь уж существенно: результаты в обоих случаях приблизительно одинаковы. При использовании интерфейсов все оказывается несколько иначе.

У абстрактных классов могут быть как абстрактные члены (у которых отсутствует тело программы и которые *обязательно* должны реализоваться в производных классах, если производный класс сам, в свою очередь, не является абстрактным), так и неабстрактные члены (которые обладают телом программы и могут быть виртуальными, т. е. *могут* переопределяться в производных классах). С другой стороны, *все* члены интерфейса должны быть реализованы в классе, который использует данный интерфейс,— ни у одного из них нет тела программы. Кроме того, все члены интерфейса являются по определению общими (поскольку предназначаются для внешнего использования), а члены абстрактного класса могут быть и частными (когда доступ к защищенным внутренним членам осуществляется исключительно из кода, находящегося внутри приложения, или из производного класса). Помимо этого, интерфейсы не могут иметь полей, конструкторов, деструкторов, статических членов и констант.

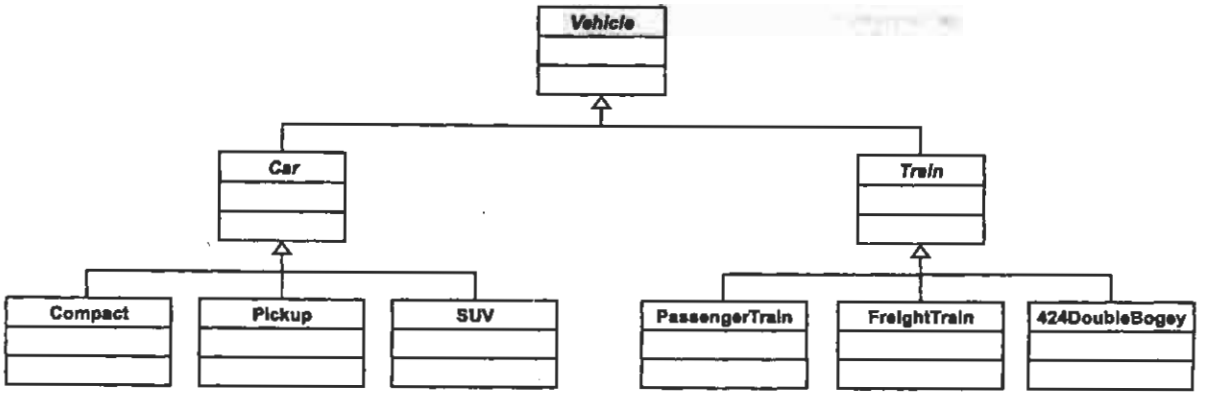
Это указывает на то, что эти два типа предназначаются для совершенно различных целей. Абстрактные классы используются в качестве базовых классов семейств тех объектов, которые обладают общими основными характеристиками, такими как цели, для которых они предназначены, и структура. Интерфейсы используются классами, отличия между которыми могут лежать на гораздо более фундаментальном уровне, но которым при этом может требоваться выполнение некоторых однаковых действий.



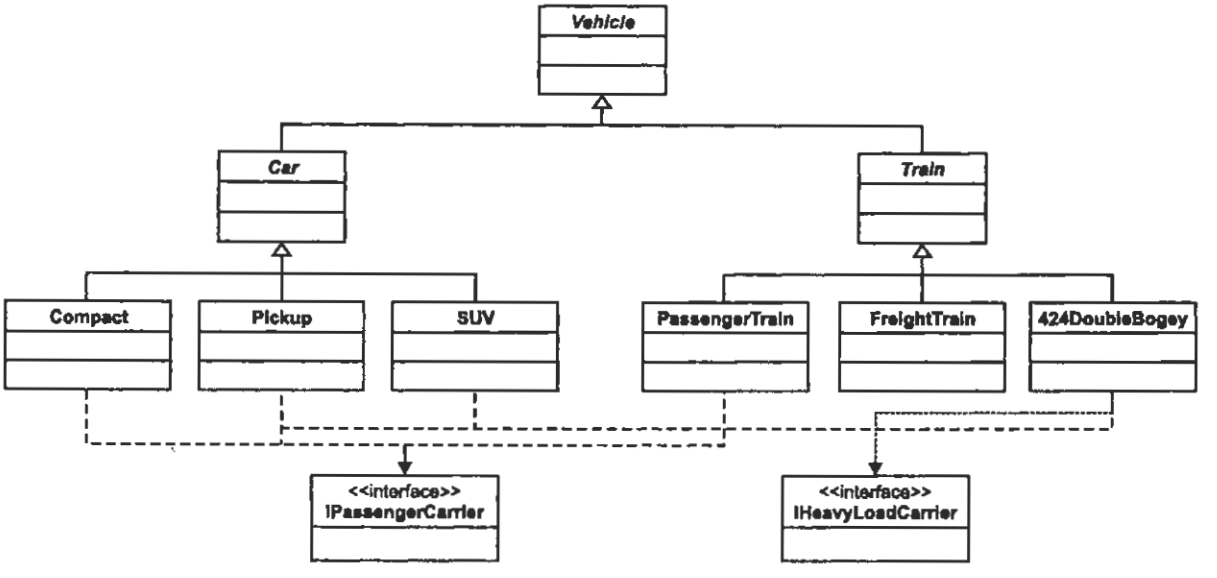
В качестве примера представьте себе семейство объектов, представляющих поезда (см. рис. слева). Базовый класс — `Train` (поезд) — содержит основное определение поезда, куда включается, например, размер колес и тип двигателя (который может быть паровым, дизельным и т. д.). Этот класс, однако, является абстрактным, поскольку в природе не существует такой вещи, как "обобщенный" поезд. Для создания "конкретного" поезда нужно задать характеристики, присущие данному поезду. Это, в свою очередь, требует создания производных классов вроде `PassengerTrain` (пассажирский поезд), `FreightTrain` (грузовой поезд) и `424DoubleBogey` (424 колесные пары).

Аналогичным образом может быть определено семейство автомобилей: с помощью базового класса `Car` (автомобиль) и производных от него классов `Compact` (компакт), `SUV` и `PickUp` (пикап). `Car` и `Train` могут даже быть производными от одного общего базового класса (`Vehicle` — средство передвижения), как показано на следующем ниже рисунке.

Некоторые из этих классов на более низких уровнях иерархии могут обладать одинаковыми характеристиками, которые определяются их предназначением, а не тем, производными от какого класса они являются. Например, `PassengerTrain`,



`Compact`, `SUV` и `PickUp` могут перевозить пассажиров, поэтому у них может быть интерфейс `IPassengerCarrier` (перевозчик пассажиров). `FreightTrain` и `PickUp` могут осуществлять перевозку тяжелых грузов, поэтому они могут обладать интерфейсом `IHeavyLoadCarrier` (перевозчик тяжелых грузов).



Создавая таким образом систему объектов, прежде чем начнем заниматься присущей каждому объекту спецификой, мы получаем возможность четко определить, в каких ситуациях использование абстрактных классов оказывается более предпочтительным, чем использование интерфейсов, и наоборот. Результатов, полученных в этом примере, нельзя достигнуть ни при использовании только интерфейсов, ни при использовании только абстрактного наследования.

Типы структур

В предыдущей главе мы обращали внимание на то, что структуры и классы очень похожи, хотя структуры представляют собой значимые типы, а классы — ссылочные типы. Что же это может означать на самом деле? Самый простой способ разобраться в этом — рассмотреть некоторый пример.

Практикум: сравнение классов и структур

1. Создайте новое консольное приложение с именем `Ch09Ex03` в директории `C:\BegCSharp\Chapter9`.

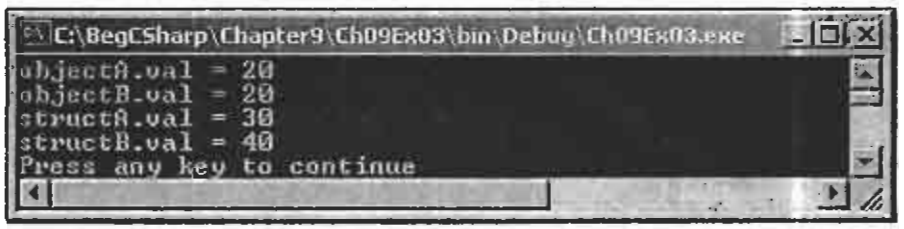
2. Измените код следующим образом:

```
namespace Ch09Ex03
{
   class MyClass
   {
      public int val;
   }

   struct myStruct
   {
      public int val;
   }

   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      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.ReadLine();
      }
   }
}
```

3. Запустите приложение:



Как это работает

В этом приложении содержатся два определения типов, одно из которых описывает структуру с именем `myStruct`, обладающую единственным общим полем типа `int` с именем `val`, а второе описывает класс с именем `MyClass`, который содержит аналогичное поле (мы будем рассматривать члены классов, такие как поля, в следующей главе, а пока вам достаточно знать, что синтаксис в данном случае

используется точно такой же). Далее мы выполняем одинаковые операции над экземплярами обоих типов:

- Объявляем переменную данного типа.
- Создаем новый экземпляр переменной данного типа.
- Объявляем вторую переменную данного типа.
- Присваиваем первую переменную второй переменной.
- Присваиваем значение полю `val` первого экземпляра переменной.
- Присваиваем значение полю `val` второго экземпляра переменной.
- Выводим значение поля `val` обеих переменных.

Хотя мы выполняем над переменными разных типов одни и те же операции, окончательные результаты оказываются различными. При выводе значений поля `val` обнаруживается, что обе переменные типа `object` обладают одинаковым значением, в то время как переменные типа структуры имеют различающиеся значения.

Итак, что же произошло?

Объекты — это *ссылочные* типы. Когда мы присваиваем объект некоторой переменной, мы на самом деле присваиваем этой переменной **указатель**, который ссылается на этот объект. С точки зрения программы указатель — это некий адрес в памяти. В данном случае это адрес, по которому располагается объект. Когда мы присваиваем ссылку на первый объект второй переменной типа `MyClass`, то мы, по сути, копируем этот адрес. Это означает, что обе переменных содержат ссылки на *один и тот же объект*.

Структуры — это *значимые* типы. В переменных не хранится никакие ссылок на структуру, вместо этого в них хранится сама структура. Поэтому, когда одна структура присваивается другой переменной типа `myStruct`, происходит копирование всей информации из одной структуры в другую. Это поведение идентично тому, которое мы наблюдали ранее для переменных простых типов, таких как тип `int`. Поэтому в качестве конечного результата мы получаем две переменные, содержащие *разные* структуры.

Весь механизм использования указателей скрыт от нас в управляемом коде, что позволяет этому коду быть существенно проще. На самом деле в C# существует возможность получить доступ к операциям более низкого уровня, таким как манипуляции над указателями, воспользовавшись "небезопасным" кодом, однако эту сложную тему мы здесь рассматривать не будем.

Сравнение неглубокого и глубокого копирования

Копирование объектов из одной переменной в другую по значению, а не по ссылке (т. е. копирование тем же способом, который используется при копировании структур) иногда оказывается делом весьма сложным. Поскольку конкретный объект может содержать большое количество ссылок на другие объекты — например, в виде своих полей, этот процесс может потребовать выполнения очень большого количества действий. Простое копирование каждого члена одного объекта в другой может не сработать, потому что некоторые из этих членов сами по себе могут иметь ссылочные типы.

В .NET Framework все это учитывается. Почленное копирование простых объектов достигается за счет использования метода `MemberwiseClone()`, который наследуется от `System.Object`. Данный метод является защищенным, однако не

составляет никакого труда определить общий метод на объекте, который его использует. Копированне, которое обеспечивается этим методом, известно под названием **неглубокого** копнрования — в том смысле, что при его осуществлении не приннмаются во внимание те члены, которые представляют собой ссылочные типы. Это означает, что ссылочные члены во вновь создаваемом объекте будут ссылаться на те же самые объекты, на которые ссылаются соответствующне члены в исходном объекте; во многнх случаях это не совсем идеальный вариант. Если нам требуется создать новые экземпляры рассматрнваемых полей и скопировать туда значення вместо ссылок, то мы должны выполнить так называемое глубокое копирование.

Существует интерфейс, который мы можем реализовать н который позволяет выполнить эту процедуру стандартным образом: `ICloneable`. Если мы собираемся его применить, нам необходимо реализовать единственный метод, содержащийся в нем,— `Clone()`. Этот метод возвращает значение типа `System.Object`. Мы можем использовать совершенно произвольные способы обработки для получения этого объекта, реализуя тело этого метода так, как нам требуется. Другими словами, мы можем реализовывать процедуру глубокого копирования в зависимости от нашего желания (например, мы при необходимости можем реализовать и процедуру неглубокого копирования).

Более детально этот вопрос будет рассматриваться в главе 11.

Итоги

В этой главе вы познакомились с определением классов и интерфейсов в C#, что позволило перевестн теорию, изложенную в предшествующей главе, в более осязаемую форму. Вы узнали, какой синтаксис в C# необходим для основных объявлений; какие ключевые слова могут быть использованы для определения уровней доступа; каким образом могут наследоваться интерфейсы и другие классы; как можно определять абстрактные и изолированные классы для того, чтобы управлять этим наследованием; а также познакомились с определеннем конструкторов и деструкторов.

В нескольких разделах вы встретились с классом `System.Object` — корневым базовым классом *всех* определяемых нами классов. Он предоставляет несколько методов, которые можно использовать, причем некоторые из них являются виртуальными, что позволяет переопределять их реализацию. Этот класс позволяет также рассматривать экземпляр произвольного объекта как экземпляр данного типа, что дает возможность использовать полиморфизм для любых объектов.

Также были рассмотрены некоторые из имеющихся в VS инструментов, предназначенных для разработок с использованием ООП, среди которых окно Class View, окно Object Browser и простой способ добавления новых классов в проект. В качестве расширения такой "мультифайловой" концепции вы познакомились со способами создания модулей, которые не могут быть выполнены, но в которых содержатся определения классов, используемые в других проектах.

Часть главы была посвящена изучению абстрактных классов и интерфейсов, в том числе рассмотрению сходства и отличия между ними, а также ситуацнй, в которых лучше задействовать одно нли другое.

В заключение мы вернулись к обсуждению ссылочных и значимых типов, несколько более подробно рассмотрев структуры (значимый тип, эквивалентный объектам). Это привело к обсуждению неглубокого и глубокого копнрования объектов — темы, к которой мы вернемся позднее.

В следующей главе мы рассмотрим определение членов класса, таких как свойства и методы, что позволит нам поднять ООП в C# на уровень, необходимый для создания реальных приложений.

Упражнения

1. Что неправильно в следующем коде?

```
public sealed class MyClass
{
   // члены класса
}

public class myDerivedClass : MyClass
{
   // члены класса
}
```

2. Как можно описать класс, который нельзя создать?

3. Почему классы, которые нельзя создать, все равно оказываются полезными? Каким образом можно их использовать?

4. Напишите код в проекте по созданию библиотеки классов под названием `Vehicles`, который реализовывал бы семейство объектов `Vehicle`, обсуждавшееся в разделе, посвященном сравнению интерфейсов и абстрактных классов. Требуется реализовать девять объектов и два интерфейса.

5. Создайте проект консольного приложения `Traffic`, который бы ссылался на `Vehicles.dll`, созданный в предыдущем упражнении. Предусмотрите функцию с именем `AddPassenger()`, которая могла бы принимать любой объект, где реализован интерфейс `IPassengerCarrier`. Чтобы убедиться в том, что программа работает, предусмотрите вызов этой функции с экземплярами каждого объекта, поддерживающего этот интерфейс, после чего вызовите для каждого объекта метод `ToString()`, наследуемый от `System.Object`, и выведите результат его работы на экран.

10

Определение членов классов

В данной главе мы продолжим обсуждение способов определения классов в C#. Теперь мы переходим к рассмотрению того, что необходимо для определения членов классов, а именно полей, свойств и методов.

Мы начнем с того, что познакомимся с кодом, требующимся для определения каждого из этих членов, а также рассмотрим создание необходимой структуры кода с помощью соответствующего мастера (wizard) VS. Мы также узнаем, каким образом можно быстро изменять члены, редактируя их свойства.

После того как вы познакомитесь с основами определения членов, мы перейдем к рассмотрению более сложных способов их использования: перевода членов базового класса в скрытое состояние, обращения к переопределенным членам базового класса и вложенных определений типов.

В заключение мы применим эту теорию на практике и создадим библиотеку классов, которую можно будет постепенно достраивать и использовать в последующих главах.

Определение членов

В рамках определения класса должны находиться определения всех членов данного класса, включая поля, методы и свойства. У каждого члена должен быть свой уровень доступа, который во всех случаях описывается одним из следующих ключевых слов:

- `public` (общий) — член, доступный из любого кода.
- `private` (частный) — член, доступный только из того кода, который является составной частью данного класса (значение по умолчанию, которое используется в случае отсутствия ключевого слова).
- `internal` (внутренний) — член, доступный только из кода, находящегося внутри проекта (модуля), в котором он определен.
- `protected` (защищенный) — член, доступный только из кода, являющегося составной частью данного класса или класса, производного от него.

Последние два ключевых слова могут использоваться совместно, т. е. член можно описать как `protected internal`. Такие члены доступны только из производных классов, описанных в рамках проекта (или, точнее, модуля; модули будут рассматриваться в главе 21).

Поля, методы и свойства могут быть описаны также посредством ключевого слова `static` (статические): в этом случае они будут статическими членами данного класса, а не экземпляров объектов, что подробно обсуждалось в главе 8.

Определение полей

Поля определяются с помощью обычного формата объявления переменной (с дополнительной возможностью инициализации) и с использованием модификаторов, обсуждавшихся выше. Например:

```
class MyClass
{
  public int MyInt;
}
```

Для имен общих полей в .NET Framework применяется система регистров PascalCasing, а не camelCasing, и в данной книге будет использоваться такой же подход. Именно поэтому поле, описанное выше, названо `MyInt`*, а не* `myInt`*. Это всего лишь рекомендуемая схема использования регистров, но она кажется автору наиболее осмысленной. Для частных полей не существует никаких рекомендаций, такие поля обычно именуются с использованием системы camelCasing.*

Для полей также может использоваться ключевое слово `readonly` (только чтение), что означает, что значение этого поля может быть задано только в процессе выполнения конструктора или при начальном присваивании. Например:

```
class MyClass
{
  public readonly int MyInt = 17;
}
```

Как было отмечено во введении к этой главе, поля могут объявляться статическими с помощью ключевого слова `static`. Например:

```
class MyClass
{
  public static int MyInt;
}
```

Доступ к статическим полям может осуществляться через класс, в котором они описаны (для вышеприведенного примера — `MyClass.MyInt`), но не через экземпляры объектов данного класса.

Кроме всего этого, существует возможность использования ключевого слова `const` (константа). Члены, описанные как `const`, являются статическими по определению, поэтому в данном случае модификатор `static` не требуется (более того, его использование является ошибочным).

Определение методов

Для описания методов используется стандартный формат функций и необязательный модификатор `static`. Например:

```
class MyClass
{
  public string GetString()
  {
    return "Это строка.";
  }
}
```

При именовании методов в .NET Framework, как и при именовании полей, используется система PascalCasing, а не camelCasing.

Заметьте, что если при описании метода использовано ключевое слово `static`, то этот метод будет доступен только через класс, но не через экземпляр объекта.

При определении метода также можно использовать следующие ключевые слова:

- `virtual` (виртуальный) — метод может быть переопределен.
- `abstract` (абстрактный) — метод должен быть переопределен (допускается только в абстрактных классах).
- `override` (переопределить) — метод переопределяет метод базового класса (это ключевое слово должно использоваться в том случае, если метод переопределяется).
- `extern` (внешний) — определение метода находится где-либо в другом месте.

Следующий пример демонстрирует переопределение метода:

```
public class MyBaseClass
{
   public virtual void DoSomething()
   {
      // Базовая реализация
   }
}

public class MyDerivedClass : MyBaseClass
{
   public override void DoSomething()
   {
      // Реализация метода в производном классе,
      // она переопределяет базовую реализацию
   }
}
```

При использовании модификатора `override` может применяться также модификатор `sealed`, который указывает на то, что никаких дальнейших модификаций этого метода в производных классах не допускается, т. е. метод не может переопределяться в производных классах. Например:

```
public class MyDerivedClass : MyBaseClass
{
   public override sealed void DoSomething()
   {
      // Реализация метода в производном классе,
      // она переопределяет базовую реализацию
   }
}
```

Использование модификатора `extern` позволяет использовать реализацию метода, внешнюю по отношению к данному проекту. Это сложная тема, и мы здесь не будем в нее углубляться.

Определение свойств

Свойства определяются аналогично полям, но с ними все оказывается не так просто. Как обсуждалось ранее, свойства сложнее полей, поскольку они допускают дополнительную обработку перед изменением своего состояния. Это достигается

за счет того, что они обладают двумя блоками, напоминающими функции, один из которых предназначается для получения значения свойства, а второй — для задания значения свойства.

Оба этих блока, определяемых посредством ключевых слов `get` (получи) и `set` (установи), могут использоваться для управления уровнем доступа к данному свойству. Существует возможность опустить тот или иной блок для создания свойства, которое будет использоваться в режиме "только чтение" или "только запись" (причем отсутствие блока `get` определяет "только запись", а отсутствие блока `set` определяет режим доступа "только чтение"). Это все, естественно, применимо только ко внешнему коду, поскольку любой код, находящийся внутри класса, будет иметь доступ ко всем тем данным, к которым имеет доступ код, находящийся в этих блоках. Для того чтобы получить корректное свойство, необходимо включить в его описание хотя бы один из этих двух блоков (свойство, которое нельзя ни считать, ни изменить, едва ли может принести хоть какую-нибудь пользу).

Основная структура свойства включает стандартное ключевое слово — модификатор доступа (`public`, `private` и т. д.), за которым следует имя типа, имя свойства и один или оба блока `set` и `get`, в которых содержится код обработки свойства. Например:

```
public int MyIntProp
{
   get
   {
      // Код для получения значения свойства
   }
   set
   {
      // Код для задания значения свойства
   }
}
```

Имена общих свойств в .NET Framework также строятся на основе регистров PascalCasing, а не camelCasing, и в этой книге мы будем использовать для них — так же, как для полей и методов — именно эту систему.

Первая строка этого определения очень напоминает определение поля. Отличие заключается в том, что в конце строки отсутствует точка с запятой; вместо нее там размещаются вложенные блоки `set` и `get`.

Блоки `get` должны обладать возвращаемым значением, имеющим тот же тип, что и само свойство. Простые свойства очень часто привязывают к единственному частному полю, которое управляет доступом к этому свойству; в этом случае блок `get` может возвращать значение этого поля напрямую. Например:

```
// Поле, используемое свойством
private int myInt;

// Свойство
public int MyIntProp
{
   get
   {
      return myInt;
   }
   set
   {
      // Код для задания свойства
   }
}
```

Заметьте, что внешний код *не будет иметь* непосредственного доступа к полю `myInt` из-за его уровня доступа (оно описано как частное). Напротив, такой код *должен* использовать свойство для того, чтобы получить доступ к полю.

Фуикция `set` присванвает полю значенне аналогичным образом. В этом случае можно воспользоваться ключевым словом `value`, чтобы сослаться на значение, полученное от пользователя:

```
// Поле, используемое свойством
private int myInt;

// Свойство
public int MyIntProp
{
   get
   {
      return myInt;
   }
   set
   {
      myInt = value;
   }
}
```

Зиачение `value` имеет такой же тип, что и свойство, поэтому, если используется тот же тип, что н тип поля, не приходится беспокоиться о приведении типов.

Это простое свойство не только защищает поле `myInt` от непосредственного доступа. Истинная мощь свойств проявляется, когда требуется несколько больший контроль над происходящим. Например, мы можем реализовать блок `set` следующим образом:

```
set
{
   if (value >= 0 && value <= 10)
      myInt = value;
}
```

В данном случае мы изменяем `myInt`, только если значение свойства находится между 0 и 10. В подобных ситуациях необходимо решить: что предпринять, если использовано недопустимое значение? Здесь имеются четыре возможности:

- Ничего не предпринимать (как в коде, приведенном выше).
- Присвоить полю значение по умолчанию.
- Продолжать как ни в чем ни бывало, но зарегистрировать событие для последующего анализа.
- Сгенерировать исключительную ситуацию.

Вообще говоря, предпочтительными являются две последние возможности, а выбор между ними зависит от того, каким образом будет использоваться данный класс и в какой степени следует передать контроль пользователям класса. Генерирование исключительной ситуации дает пользователям очень большие возможности контроля, позволяет получать информацию о том, что происходит, и соответственно реагировать. Для этого можно воспользоваться стандартной исключительной ситуацией `System`, например:

```
set
{
   if (value >= 0 && value <= 10)
      myInt = value;
```

```
        else
            throw (new ArgumentOutOfRangeException("MyIntProp", value,
                        "Значение свойства MyIntProp должно лежать в
                         диапазоне между 0 и 10."));
    }
```

Эту ситуацию можно обработать, применив конструкцию `try...catch...finally` в коде, который использует данное свойство, как это было описано в главе 7.

Регистрация данных, возможно, в текстовом файле, может оказаться полезной для производственных программ, в которых никаких проблем возникать не должно. Эти данные позволяют разработчикам проверять работоспособность программ и, при необходимости, исправлять обнаруженные ошибки.

Свойства, так же как и методы, могут использовать ключевые слова `virtual`, `override` и `abstract`, т. е. те ключевые слова, использование которых для полей оказывается недопустимым.

Практикум: использование полей, методов и свойств

1. Создайте новый проект консольного приложения с именем `Ch10Ex01` в директории `C:\BegCSharp\Chapter10`.
2. С помощью VS добавьте новый класс с именем `MyClass` в файл `MyClass.cs`.
3. Измените код в `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."));
// Значение, присваиваемое Val, должно лежать в диапазоне между 0 и 10. //
        }
    }

    public override string ToString()
    {
        return "Name: " + Name + "\nVal: " + Val;
    }

    private MyClass() : this("Default Name")
    {
    }

    public MyClass(string newName)
    {
        Name = newName;
        intVal = 0;
    }
}
```

4. Измените код в Class1.cs следующим образом:

```
static void Main(string[] args)
{
   Console.WriteLine("Creating object myObj...");
   MyClass myObj = new MyClass("My Object");
   Console.WriteLine("myObj created.");
   for (int i = -1; i <= 0; i++)
   {
      try
      {
         Console.WriteLine("\nAttempting to assign {0} to myObj.Val...", i);
         myObj.Val = i;
         Console.WriteLine("Value {0} assigned to myObj.Val.", myObj.Val);
      }
      catch (Exception e)
      {
         Console.WriteLine("Exception {0} thrown.", e.GetType().FullName);
         Console.WriteLine("Message:\n\"{0}\"", e.Message);
      }
   }
   Console.WriteLine("\nOutputting myObj.ToString()...");
   Console.WriteLine(myObj.ToString());
   Console.WriteLine("myObj.ToString() Output.");
}
```



5. Запустите приложение (см. рис. слева).

Как это работает

Код в Main() создает и использует экземпляр класса MyClass, который определен в MyClass.cs. Создание экземпляра данного класса должно осуществляться с помощью конструктора не по умолчанию, поскольку конструктор по умолчанию класса MyClass является частным:

```
private MyClass() : this("Default Name")
{
}
```

Обратите внимание, что используется this("Default Name") — для гарантированного присвоения Name какого-либо значения при обращении к конструктору. Последнее возможно, если данный класс используется для создания нового класса. Отсутствие значения у поля Name в дальнейшем может привести к возникновению ошибок.

Используемый не по умолчанию конструктор присваивает значения описанному как readonly полю name (это присваивание можно осуществить либо при объявлении поля, либо с помощью конструктора) и частному полю intVal.

Далее Main() пытается выполнить два присваивания свойству Val объекта myObj (он является экземпляром класса MyClass). Для присваивания значений –1 и 0 используются два прохода цикла for, а для обнаружения возможных исключительных ситуаций применяется конструкция try...catch. Когда свойству присваивается –1, возникает исключительная ситуация System.ArgumentOutOfRangeException и код, находящийся в блоке catch, выводит информацию о ней в окно консоли. При

следующем проходе цикла свойству `Val` присваивается значение 0, а затем через это свойство значение присваивается частному полю `intVal`.

В заключение для вывода отформатированной строки, представляющей содержимое объекта, используется переопределенный метод `ToString()`:

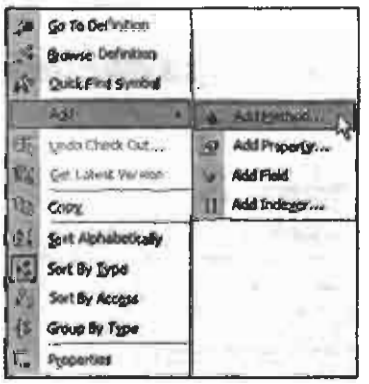
```
public override string ToString()
{
   return "Name: " + Name + "\nVal: " + Val;
}
```

Этот метод должен быть объявлен с использованием ключевого слова `override`, поскольку названный метод переопределяет виртуальный метод `ToString()` базового класса `System.Object`. Код в данном случае использует непосредственно свойство `Val`, а не частное поле `intVal`. Нет никаких причин, по которым мы не могли бы использовать свойства внутри класса подобным образом, хотя в этом случае может возникать небольшое замедление работы программы (настолько небольшое, что мы вряд ли сумеем его обнаружить). Использование свойства, кроме того, позволяет осуществлять контроль за допустимостью значений, присущий использованию свойств, что также может оказаться полезным для кода, находящегося внутри класса.

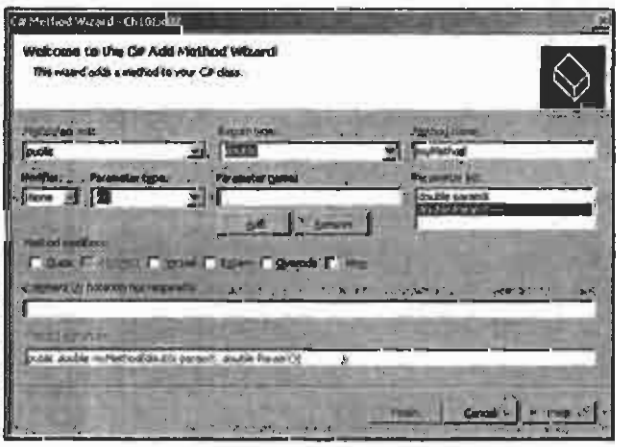
Мастера VS для работы с членами

В предыдущей главе вы познакомились с использованием вспомогательных программ в VS для создания нового класса, укомплектованного собственным `.cs`-файлом. Кроме них, существует еще некоторое количество таких программ (известных под названием мастеров), которые могут применяться для внесения изменений в классы, а именно: мастера для добавления классам свойств, методов и полей. К рассмотрению этих мастеров мы и приступаем.

Доступ ко всем мастерам для работы с членами осуществляется одним и тем же способом. В окне Class View, в котором выводятся описанные в проектах классы вместе со своими членами, необходимо щелкнуть правой кнопкой мыши по соответствующему классу и выбрать из раскрывшегося меню опцию Add (см. рис. справа).



В этом меню имеется четыре возможности, но мы в настоящий момент будем рассматривать только три из них, поскольку индексаторы являются предметом изучения следующей главы.



Мастер Add Method (добавление методов)

Данный мастер — кто бы мог подумать! — позволяет добавлять методы в класс. Это достигается с помощью диалогового окна Method Wizard, представленного на рисунке слева (на рисунке некоторые его поля уже заполнены).

Названное окно обеспечивает доступ ко всем возможностям методов. Мы получаем средство, позволяющее изменять уровень доступа

с помощью раскрывающегося поля Method access (доступ к методу), тип возвращаемого значения — с помощью раскрывающегося поля Return type (тип возвращаемого значения; кроме того, существует возможность ввести в это поле наше собственное название типа), имя — с помощью раскрывающегося поля Method name (имя метода), а также другие модификаторы — с помощью Method modifiers (модификаторы метода). Поле модификаторов метода допускает использование только корректных сочетаний, и в абстрактных классах возможны только абстрактные методы. Мы можем также добавлять параметры, используя для этой цели поля Parameter type (тип параметра) и Parameter name (имя параметра). Поле Modifier (модификатор) позволяет задавать тип параметра: обычный (None), `ref` или `out`. Добавленные параметры появляются в поле Parameter list (список параметров), когда мы нажимаем кнопку Add (добавить); исключать параметры из списка можно с помощью кнопки Remove (удалить).

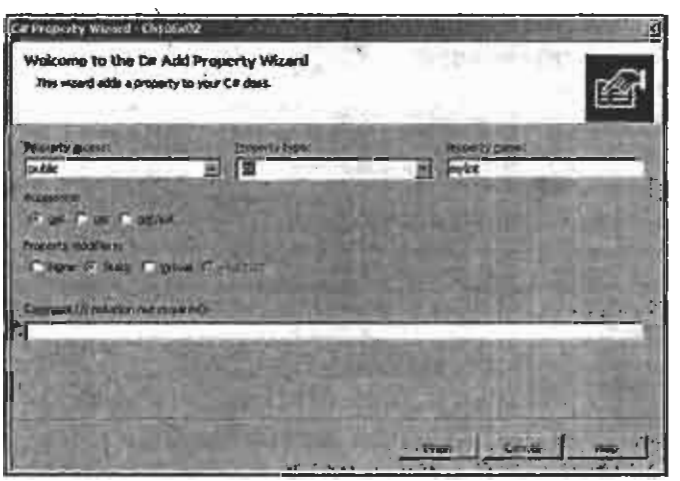
Это диалоговое окно позволяет ввести комментарий, помещаемый перед определением метода, в поле Comment (комментарий), а также посмотреть код, который будет сгенерирован, в окне Method signature (сигнатура метода).

Нажатие кнопки Finish (завершение) при тех установках, которые изображены на рисунке, приведет к добавлению следующего кода в наш класс:

```
public double myMethod(double paramX, double ParamY)
{
    return 0;
}
```

Совершенно очевидно, что мастер оказывается не в состоянии реализовать сам метод за нас, однако он создает базовую структуру метода и, безусловно, существенно снижает количество ошибок при вводе текста!

Мастер Add Property (добавление свойств)



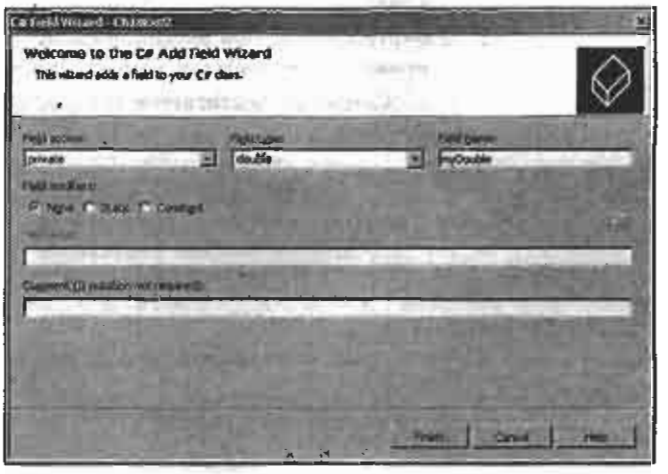
Мастер Add Property использует диалоговое окно, показанное на рисунке слева (некоторые поля нами уже заполнены).

Мы выбираем тип доступа с помощью раскрывающегося поля Property access (доступ к свойству), тип свойства — с помощью поля Property type (тип свойства), вводим имя в поле Property name (имя свойства), определяем, какие блоки нам необходимы — `get`, `set` или одновременно и `get`, и `set`, а также по желанию добавляем модификаторы и комментарии. Нажатие кнопки Finish (завершение) при изображенных на рисунке установках создает следующий код:

```
public static int myInt
{
    get
    {
        return 0;
    }
}
```

Обратите внимание, что окончательная реализация свойства возлагается на нас (в случае простых свойств она сводится к установлению соответствия между свойством и полем). Однако теперь имеется возможность использовать базовую структуру свойства.

Мастер Add Field (добавление поля)



Наконец, рассмотрим мастер Add Field (см. рис. справа). Это диалоговое окно оказывается еще проще.

Оно в основном схоже с диалоговым окном для добавления свойств. Исключение составляет отсутствие возможности выбирать блоки `get` и `set` и использовать ключевые слова `virtual` и `abstract`. Однако можно определить поле как константу и воспользоваться Field value (значение поля) для задания его значения.

Код при этом генерируется простой и самоописательный:

```
private double myDouble;
```



Свойства членов

Последняя из основных тем, которую необходимо рассмотреть,— изменение свойств членов с помощью окна Properties. После того как мы выбрали член в окне Class View, мы получаем возможность просматривать его свойства в окне Properties (см. рис. слева).

Из этого окна мы можем непосредственно изменять многие свойства, напрнмер, уровень доступности, посредством свойства Access (доступ) (оно выделено на рнсунке). В этом случае код будет модифицирован автоматнчески — н не придется ннчего набирать на клавиатуре.

Обратите внимание, что свойство CanOverride (может переопределяться) позволяет определнть, является ли член виртуальным, а IsShared (с разделением доступа) — является ли он статическим.

Некоторые дополнительные темы, касающиеся членов класса

Теперь, когда вы познакомилнсь с основами определения членов, самое время перейти к рассмотрению более сложных тем. Здесь будут рассмотрены следующие из них:

- Сокрытие методов базового класса
- Вызов переопределенных или скрытых методов базового класса
- Вложеиные определения типов

Сокрытие методов базового класса

При наследовании некоторого (не абстрактного) члена базового класса наследуется также и его реализация. Если наследуемый члеи является виртуальным, то имеется возможность переопределить его реализацию, воспользовавшись ключевым словом override. Независимо от того, является ли наследуемый член виртуальным, при необходимости можно скрыть его реализацию. Это полезио, например, если наследуемый общий член работает не так, как требуется.

Сокрытие достигается использованием следующего кода:

```
public class MyBaseClass
{
   public void DoSomething()
   {
      // Базовая реализация
   }
}

public class MyDerivedClass : MyBaseClass
{
   public void DoSomething()
   {
      // Реализация в производном классе, скрывающая базовую реализацию
   }
}
```

Хотя такой код будет работать нормально, он сгенерирует предупреждение о том, что в нем скрывается член базового класса. Это дает возможность исправить положенне дел, еслн мы случайно скрылн член, который на самом деле желаем использовать. Если же этот член действительно требуется скрыть, то об этом можно сказать явно, воспользовавшнсь ключевым словом new:

```
public class MyDerivedClass : MyBaseClass
{
   new public void DoSomething()
   {
      // Реализация в производном классе, скрывающая базовую реализацию
   }
}
```

Этот код будет работать точно так же, только без выдачн предупреждения.

На данном этапе нам представляется важным отметить, в чем заключается разница между скрытыми и переопределенными членами базового класса. Рассмотрнм следующий код:

```
public class MyBaseClass
{
   public virtual void DoSomething()
   {
      Console.WriteLine("Базовая реализация");
   }
}

public class MyDerivedClass : MyBaseClass
{
   public override void DoSomething()
   {
      Console.WriteLine("Производная реализация");
   }
}
```

В данном случае переопределенный метод заменяет собой реализацию базового класса, так что в следующем коде будет использоваться новая версия, несмотря на то, что обращение к методу происходит через тип базового класса (с использованием полиморфизма):

```
MyDerivedClass myObj = new MyDerivedClass();
MyBaseClass myBaseObj;
myBaseObj = myObj;
myBaseObj.DoSomething();
```

В данном случае выходной поток будет следующим:

```
Производная реализация
```

В качестве альтернативы мы можем скрыть метод базового класса с помощью следующего кода:

```
public class MyBaseClass
{
   public virtual void DoSomething()
   {
      Console.WriteLine("Базовая реализация");
   }
}

public class MyDerivedClass : MyBaseClass
{
   new public void DoSomething()
   {
      Console.WriteLine("Производная реализация");
   }
}
```

Метод базового класса при этом совершенно не обязательно должен быть виртуальным, однако от этого ничего не изменится, и приведенный здесь код отличается от предыдущего варианта только одной строкой. Результат выполнения этого кода как при виртуальном методе, так и в противном случае будет следующим:

```
Базовая реализация
```

Несмотря на то что базовая реализация является скрытой, к ней, как и раньше, имеется доступ через базовый класс.

Вызов переопределенных или скрытых методов базового класса

Независимо от того, переопределяем ли мы некоторый член или делаем его скрытым, по-прежнему остается возможность доступа к члену базового класса внутри данного класса. Существует множество различных ситуаций, когда это оказывается полезным, например:

- Если требуется скрыть наследуемый общий член от пользователей производного класса, но необходимо иметь доступ к его функциональным возможностям внутри класса.
- Если требуется что-либо добавить к реализации наследуемого виртуального члена, а не просто заменить ее на новую переопределенную реализацию.

Для достижения подобного эффекта можно воспользоваться ключевым словом `base`, которое указывает на реализацию базового класса, содержащуюся внутри

производного класса (аналогично тому, как оно используется в управляющих конструкторах, с которыми вы познакомились в предыдущей главе). Например:

```
public class MyBaseClass
{
   public virtual void DoSomething()
   {
      // Базовая реализация
   }
}

public class MyDerivedClass : MyBaseClass
{
   public override void DoSomething()
   {
      // Реализация в производном классе, расширяющая реализацию
      в базовом классе base.DoSomething();
      // Дополнительная реализация в производном классе
   }
}
```

В представленном коде выполняется версия `DoSomething()`, содержащаяся в классе `MyBaseClass` (базовом по отношению к `MyDerivedClass`), из которой вызывается версия `DoSomething()`, содержащаяся внутри класса `MyDerivedClass`.

Поскольку ключевое слово `base` работает с экземплярами объектов, то будет ошибкой использовать его внутри статического члена.

Ключевое слово this

Кроме ключевого слова `base`, в предыдущей главе мы также использовали ключевое слово `this`. Так же, как и `base`, `this` может применяться внутри членов класса, и так же, как и `base`, оно относится к экземпляру объекта. Экземпляр объекта, на который указывает `this`, является текущим экземпляром объекта (отсюда следует, что это ключевое слово нельзя использовать в статических членах, поскольку они не являются составной частью экземпляра объекта).

Наиболее ценной функцией ключевого слова `this` является возможность передавать некоторому методу ссылку на текущий экземпляр объекта, например:

```
public void doSomething()
{
   MyTargetClass myObj = new MyTargetClass();
   myObj.doSomethingWith(this);
}
```

В данном примере класс `MyTargetClass`, экземпляр которого создается, обладает методом `DoSomethingWith()`. Ему передается единственный параметр, имеющий тип, совместимый с тем классом, в котором содержится этот метод. Тип параметра может совпадать с типом данного класса, с типом класса, наследником которого является данный класс, может являться интерфейсом, реализованном в данном классе, или же (естественно) быть типом `System.Object`.

Вложенные определения типов

Помимо описания типов как классов в пространствах имен их можно описывать внутри других классов. В таком случае появляется возможность использовать в определениях весь спектр модификаторов доступности, а не только `public` и `internal`, а также применять ключевое слово `new` для перевода определения типа, наследуемого от базового класса, в скрытое состояние.

В следующем коде, кроме класса `MyClass`, описывается также вложенный класс с именем `myNestedClass`:

```
public class MyClass
{
   public class myNestedClass
   {
      public int nestedClassField;
   }
}
```

Если потребуется создать экземпляр класса `myNestedClass` откуда-нибудь извне класса `MyClass`, то его имя должно квалифицироваться:

```
MyClass.myNestedClass myObj = new MyClass.myNestedClass();
```

Однако вполне возможно, что мы окажемся не в состоянии выполнить такие действия, если вложенный класс объявлен как частный или обладает каким-либо другим уровнем доступности, несовместимым с кодом в той точке, в которой осуществляется попытка создания соответствующего экземпляра.

Основной причиной этого является необходимость описания классов, которые являются частными по отношению к содержащему их классу, так что никакой другой код в данном пространстве имен не будет иметь к ним доступа.

Реализация интерфейсов

Перед тем как двигаться дальше, имеет смысл более подробно рассмотреть способы определения и реализации интерфейсов. В предшествующей главе вы узнали, что интерфейсы определяются аналогично классам, для чего используется код следующего вида:

```
interface IMyInterface
{
   // члены интерфейса
}
```

Члены интерфейса определяются так же, как и члены класса, за исключением нескольких существенных отличий:

- Не допускается использование никаких модификаторов доступности (`public`, `private`, `protected` и `internal`) — все члены интерфейса неявно общие.
- Члены интерфейса не могут содержать тела программы.
- Интерфейсы не могут описывать члены в виде полей.
- При описании членов интерфейса не могут использоваться ключевые слова `static`, `virtual`, `abstract` и `sealed`.
- Члены описания типов не допускаются.

Существует, однако, возможность использовать при описании членов ключевое слово `new` в тех случаях, когда требуется перевести в скрытое состояние члены, наследуемые от базовых интерфейсов. Например:

```
interface IMyBaseInterface
{
   void DoSomething();
}
```

```
interface IMyDerivedInterface : IMyBaseInterface
{
   new void DoSomething();
}
```

Это работает точно таким же образом, как и перевод в скрытое состояние наследуемых членов класса.

В свойствах, определяемых в интерфейсах, задается, какие блоки доступа — `get` и/или `set` — допустимы для данного свойства; например:

```
interface IMyInterface
{
   int MyInt
   {
      get;
      set;
   }
}
```

В данном случае свойство `MyInt` типа `int` обладает обоими средствами доступа — и `get`, и `set`. Любое из них может быть опущено для свойства, доступ к которому является более ограниченным.

Однако обратите внимание на то, что интерфейсы не определяют, каким именно образом должно храниться свойство. Они не могут указать, например, поля, которые использовались бы для хранения данных, определяющих свойство.

Наконец, интерфейсы, как и классы, могут определяться как члены классов (но не как члены других интерфейсов, поскольку интерфейсы не могут содержать в себе определения типов).

Реализация интерфейсов в классах

Класс, в котором реализован интерфейс, должен содержать в себе реализации всех членов данного интерфейса, которые, в свою очередь, должны соответствовать указанной сигнатуре (включая соответствие указанным блокам `get` и `set`) и быть общими. При реализации членов интерфейса допускается использование ключевых слов `virtual` и `abstract`, но не `static` или `const`. Например:

```
public interface IMyInterface
{
   void DoSomething();
   void DoSomethingElse();
}

public class MyClass : IMyInterface
{
   public void IMyInterface.DoSomething()
   {
   }

   public void DoSomethingElse()
   {
   }
}
```

Интерфейсы также могут реализовываться на базовых классах, например:

```
public interface IMyInterface
{
   void DoSomething();
   void DoSomethingElse();
}
```

```
public class MyBaseClass
{
   public void DoSomething()
   {
   }
}

public class MyDerivedClass : MyBaseClass, IMyInterface
{
   public void DoSomethingElse()
   {
   }
}
```

Наследование из базового класса, который реализует данный интерфейс, означает, что этот интерфейс опосредованно поддерживается и производным классом:

```
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()
   {
   }
}
```

Как было показано выше, реализации в базовом классе полезно определять как виртуальные — для того, чтобы производные классы могли заменять эти реализации, а не переводить их в скрытое состояние. Если мы переведем метод в скрытое состояние с помощью ключевого слова `new`, вместо того чтобы переопределить его таким образом, то в этом случае метод `IMyInterface.DoSomething()` будет всегда ссылаться на версию, описанную в базовом классе, даже если с помощью этого интерфейса осуществляется доступ к производному классу.

Явная реализация членов интерфейса

Члены интерфейса также могут реализовываться в классе **явным** образом. В этом случае доступ к членам осуществляется только через интерфейс, но не через класс. Доступ к **неявным** членам, т. е. к тем, которые мы использовали в коде в предыдущем разделе, может осуществляться обоими способами.

Например, если в классе `MyClass` метод `DoSomething()` интерфейса `IMyInterface` реализован неявным образом, следующий код является допустимым:

```
MyClass myObj = new MyClass();
myObj.DoSomething();
```

Как и такой:

```
MyClass myObj = new MyClass();
IMyInterface myInt = myObj;
myInt.DoSomething();
```

Напротив, в том случае, если в классе `MyDerivedClass` метод `DoSomething()` реализован явным образом, то разрешается использовать только второй способ. Для этого потребуется записать следующий код:

```
public class MyClass : IMyInterface
{
   void IMyInterface.DoSomething()
   {
   }

   public void DoSomethingElse()
   {
   }
}
```

Здесь `DoSomething()` реализован явным образом, а `DoSomethingElse()` — неявным. Непосредственный доступ через экземпляр объекта класса `MyClass` допускается только ко второму из них.

Пример приложения

Для того чтобы проиллюстрировать некоторые приемы, описывавшиеся нами ранее, мы создадим модуль классов, который сможем достраивать и использовать в последующих главах. Этот модуль будет содержать в себе два класса:

- Класс `Card` (карта) представляет собой стандартную игральную карту какой-либо масти — треф, бубен, червей или пик — и определенного старшинства в диапазоне между тузом и королем.
- Класс `Deck` (колода) представляет собой полную колоду из 52 карт, обеспечивает возможность получать карту по ее местоположению в колоде и позволяет перетасовывать карты.

Мы также разработаем простое клиентское приложение, чтобы убедиться, что все это работает, но использовать колоду в полноценном приложении для карточных игр мы не будем — пока не будем!

Планирование приложения

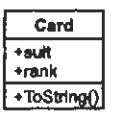
Наши классы будут содержаться в библиотеке классов этого приложения — `Ch10CardLib`. Прежде чем приступать к написанию какого бы то ни было кода, нам необходимо спланировать необходимую структуру и функциональные возможности наших классов.

Класс Card

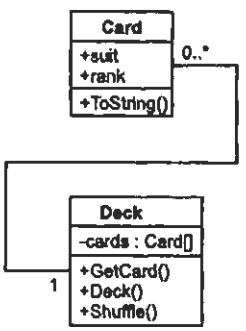
Класс `Card` в основном будет служить контейнером для двух полей, использующихся в режиме "только чтение": `suit` (масть) и `rank` (старшинство). Причина, по которой мы собираемся создавать поля, используемые в режиме "только чтение", заключается в том, что нет смысла использовать "пустую" карту и не должно

существовать возможности изменять карты после того, как они созданы. Для достижения этой цели мы опишем конструктор по умолчанию как частный и введем альтернативный конструктор, который будет создавать карту по заданным масти и старшинству.

Кроме этого, в классе `Card` будет еще переопределен метод `ToString()` класса `System.Object` таким образом, чтобы можно было простым способом получать удобоваримое строковое представление любой карты. Ради некоторого упрощения задачи для обоих этих полей — `suit` и `rank` — мы определим перечислимые типы.



Класс `Card` может быть представлен, как показано на рисунке справа.



Класс Deck

Класс `Deck` будет поддерживать 52 объекта `Card`. Для этого мы используем тип простого массива. Непосредственного доступа к этому массиву предусмотрено не будет — доступ к объектам `Card` будет осуществляться посредством метода `GetCard()` (получение карты), который будет возвращать карту по заданному индексу.

Этот класс также должен обладать методом `Shuffle()` (тасуй), который позволяет расположить карты в массиве по-новому, поэтому его можно представить, как показано на рисунке слева.

Создание библиотеки классов

Для целей настоящего примера авторы исходили из предположения, что вы уже настолько хорошо знакомы с VS, что не нуждаетесь в стандартном "Практикуме", поэтому явно последовательность необходимых действий не указана.

Оба класса, а также перечислимые типы будут содержаться в проекте по созданию библиотеки классов с именем `Ch10CardLib`. Этот проект будет состоять из двух `.cs`-файлов: `Card.cs`, в котором будет находится определение класса `Card` вместе с двумя перечислимыми типами `Suit` и `Rank`, и `Deck.cs`, в котором будет находится определение класса `Deck`.

Card.cs

В этом разделе мы подробно разберем и проработаем код для `Card.cs`. В начале, как всегда, мы включим обычный оператор `using` и объявление пространства имен:

```
using System;

namespace Ch10CardLib
{
```

Далее мы поместим определение перечислимого типа `Suit`:

```
public enum Suit
{
   Club,          //трефа//
   Diamond,       //бубна//
   Heart,         //черва//
   Spade          //пика//
}
```

Теперь нам потребуется определение перечислимого типа Rank, которое для простоты будет начинаться с представления базового типа для туза, равного 1:

```
public enum Rank
{
   Ace = 1,        //туз//
   Deuce,          //двойка//
   Three,          //тройка//
   Four,           //четверка//
   Five,           //пятерка//
   Six,            //шестерка//
   Seven,          //семерка//
   Eight,          //восьмерка//
   Nine,           //девятка//
   Ten,            //десятка//
   Jack,           //валет//
   Queen,          //дама//
   King            //король//
}
```

Оба этих типа являются общими, поскольку мы рассчитываем, что они будут использоваться извне данной библиотеки классов.

Теперь перейдем к основной части этого файла — определению класса Card:

```
public class Card
{
```

Первая часть кода в определении класса Card объявляет два поля, используемых в режиме "только чтение" и обладающих перечислимыми типами, определенными в начале данного файла:

```
public readonly Suit suit;

public readonly Rank rank;
```

Переопределенный метод ToString() оказывается очень простым благодаря использованию перечислимых типов:

```
   public override string ToString()
   {
      return "The " + rank + " of " + suit + "s";
   }
```

Включение имен полей в строку таким способом приведет просто к переносу строкового представления хранящихся значений перечислимого типа в возвращаемую строку.

Обратимся к конструкторам. Сначала — частный конструктор по умолчанию:

```
   private Card()
   {
   }
```

Следующим должен быть конструктор, который будет применяться для создания объектов класса Card; ему просто передаются параметры Suit и Rank для инициализации полей, использующихся в режиме "только чтение":

```
   public Card(Suit newSuit, Rank newRank)
   {
      suit = newSuit;
      rank = newRank;
   }
}
```

На этом код данного файла завершен.

Deck.cs

Этот файл начинается так же, как и Card.cs,— со стандартного кода:

```
using System;

namespace Ch10CardLib
{
```

В данном случае нам не требуется описывать какие-либо типы, кроме класса Deck, поэтому мы сразу переходим к нему:

```
   public class Deck
   {
```

Первым мы опишем член, представляющий собой частный массив объектов типа Card, который мы назовем cards:

```
      private Card[] cards;
```

Затем следует конструктор, который просто создает и присваивает 52 карты полю cards. Мы проходим в цикле все возможные сочетания двух перечислимых типов, используя каждое из этих сочетаний для создания соответствующей карты. В результате мы получаем поле cards, которое изначально содержит упорядоченный перечень карт:

```
      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); }
            }
         }
```

Далее мы реализуем метод GetCard(), который либо возвращает объект Card с требуемым индексом, либо генерирует исключительную ситуацию так, как мы делали это раньше:

```
      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."));
      }
```

Наконец, требуется реализовать метод Shuffle(). Он работает за счет создания временного массива карт и копирования карт из существующего массива cards в этот массив случайным образом. Основным телом метода является цикл, проходящий значения от 0 до 51. При каждом проходе генерируется случайное число в диапазоне от 0 до 51 при помощи экземпляра класса System.Random, находящегося в .NET Framework. Этот экземпляр генерирует случайные числа в диапазоне между 0 и *X* с помощью метода Next(*X*). Полученное число используется в качестве индекса того объекта Card во временном массиве, куда осуществляется запись очередной карты из массива cards.

Для учета записанных карт будет использоваться массив переменных типа bool, в который заносится значение true для каждой перепнсаиной карты. Когда мы

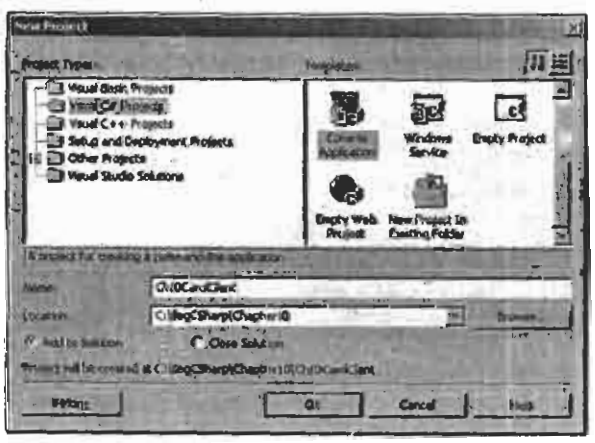
получаем случайное число, то проверяем соответствующее значение в этом массиве на предмет того, не была ли уже какая-нибудь карта записана в него в место, определяемое данным числом, и если была, то просто генерируем следующее число.

Такой способ тасования карт не является максимально эффективным, поскольку может потребоваться сгенерировать большое количество случайных чисел, прежде чем будет найдено свободное место для копирования очередной карты. Однако такой метод вполне работоспособен, а поскольку код на C# выполняется очень быстро, мы едва ли сможем заметить какую-либо задержку.

Итак, этот код имеет следующий вид:

```
        public void Shuffle()
        {
            Card[] newDeck = new Card[52];
            bool[] assigned = new bool[52];
            for (int i = 0; i < 52; i++)
            {
                int destCard = 0;
                bool foundCard = false;
                Random sourceGen = new Random();
                while (foundCard == false)
                {
                    destCard = sourceGen.Next(52);
                    if (assigned[destCard] == false)
                        foundCard = true;
                }
                assigned[destCard] = true;
                newDeck[destCard] = cards[i];
            }
            cards = newDeck;
        }
    }
}
```

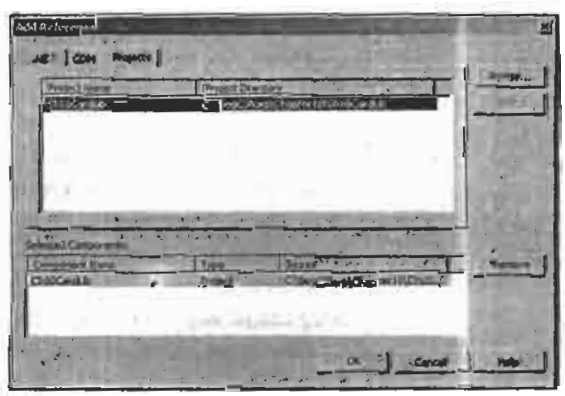
На этом код нашей библиотеки классов завершен.



Для использования только что созданной библиотеки классов из нового проекта по созданию консольного приложения необходимо всего лишь добавить ссылку на проект по созданию библиотеки классов `Ch10CardLib`. После того как проект создан, этого можно достичь с помощью вкладки Projects диалогового окна Add Reference (см. рис. справа).

Клиентское приложение для библиотеки классов

Для большей простоты мы можем добавить клиентское консольное приложение к тому решению, в котором содержится библиотека классов. Для этого достаточно выбрать при создании проекта, который назовем Ch10CardClient, опцию Add to Solution (см. рис. слева).



Выберите проект, щелкните мышью сначала по кнопке **Select**, затем по кнопке **OK**, и проект будет добавлен.

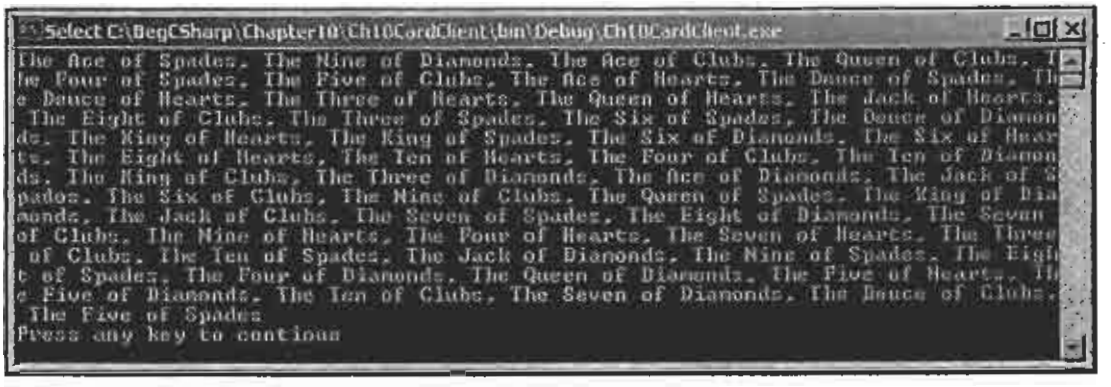
Поскольку вновь создаваемый проект является одним из двух имеющихся проектов, потребуется указать, что он является стартовым проектом решения — т. е. что именно он будет выполняться при нажатии кнопки запуска. Для этого щелкнем правой кнопкой мыши по имени проекта в окне **Solution Explorer** и выберем из меню опцию **Set as StartUp Project** (использовать в качестве стартового проекта).

Далее потребуется ввести код, который будет использовать новые классы. Для этого вполне достаточно следующего кода:

```
using System;
using Ch10CardLib;

namespace Ch10CardClient
{
   class Class1
   {
      static void Main(string[] args)
      {
         Deck myDeck = new Deck();
         myDeck.Shuffle();
         for (int i = 0; i < 52; i++)
         {
            Card tempCard = myDeck.GetCard(i);
            Console.WriteLine(tempCard.ToString());
            if (i != 51)
               Console.Write(", ");
            else
               Console.WriteLine();
         }
      }
   }
}
```

В результате будет получено следующее:



52 игральные карты располагаются в колоде случайным образом.

Мы продолжим развивать и использовать данную библиотеку классов в последующих главах.

Итоги

В этой главе мы обсудили, каким образом определяются базовые классы. Вам еще довольно много предстоит узнать, однако описанные способы уже позволяют создавать достаточно сложные приложения.

По ходу главы мы рассмотрели вопрос о том, как определяются поля, методы и свойства, обсудили различные уровни доступа и ключевые слова — модификаторы. Кроме того, вы познакомились с мастерами и некоторыми другими инструментами, которые могут вдвое ускорять процесс описания класса.

После разбора этих основополагающих тем мы очень подробно изучили вопросы наследования, познакомившись с переводом ненужных наследуемых членов в скрытое состояние посредством ключевого слова `new`, а также с тем, как можно расширять члены базового класса с помощью ключевого слова `base` вместо того, чтобы заменять их реализацию новой. Также были рассмотрены вложенные определения классов.

После этого мы уделили более пристальное внимание определению и реализации интерфейсов, включая понятия явной и неявной реализации.

В заключение была создана и применена простая библиотека классов, представляющая собой колоду игральных карт. Мы собираемся использовать эту библиотеку и дальше.

В следующей главе будут рассмотрены некоторые из наиболее сложных возможностей создания классов — такие как семейства, перегрузка операторов и глубокое копирование.

Упражнения

1. Напишите код, определяющий базовый класс `MyClass`, с помощью виртуального метода `GetString()`. Этот метод должен возвращать строку, хранимую в защищенном поле `myString`, которое доступно через общее свойство `ContainedString` в режиме "только запись".

2. Создайте класс `MyDerivedClass`, производный от класса `MyClass`. Переопределите метод `GetString()` таким образом, чтобы он возвращал строку из базового класса, используя базовую реализацию метода, но при этом добавлял бы в получаемую строку текст " (output from derived class)" ("вывод из производного класса").

3. Создайте класс с именем `MyCopyableClass`, который позволял бы получать копию самого себя с помощью метода `GetCopy()`. Этот метод должен использовать метод `MemberwiseClone()`, наследуемый от `System.Object`. Добавьте в класс какое-либо простое свойство и напишите клиентский код, который, используя этот класс, позволит убедиться, что все работает правильно.

4. Создайте клиентское консольное приложение для библиотеки `Ch10CardLib`, которое "сдает" по пять карт из перетасованной колоды, хранящейся в объекте `Deck`. В том случае, если все пять карт окажутся одной масти, оно должно вывести на экран их названия вместе с текстом "Flush!" ("Флеш!"); в противном случае оно должно заканчивать работу после сдачи 50 карт, выводя текст "No flush" ("Нет флеша").



Дополнительные сведения о классах

К настоящему моменту мы рассмотрели практически все основные приемы ООП, использующиеся в C#; однако существует несколько более сложных способов, с которыми также не мешает познакомиться. В этой главе будут обсуждаться следующие темы:

- **Семейства** — это объекты, в которых могут содержаться массивы других объектов и которые обладают функциональными возможностями для организации доступа к ним.
- **Перегрузка операторов** — процедура конфигурирования классов с тем, чтобы иметь возможность применять такие операторы, как, например, +, для выполнения операций над экземплярами данного класса.
- **Усложненное преобразование** — некоторые дополнительные возможности по преобразованию типов в C#.
- **Глубокое копирование** — это процедура копирования, гарантирующая, что создаваемые объекты не содержат ссылок на данные, хранящиеся в исходном объекте.
- **Исключительные ситуации пользователя** — создание программистом собственных исключительных ситуаций для того, чтобы передавать коду, перехватившему данную исключительную ситуацию, некоторую дополнительную информацию.

Семейства

В главе 5 вы познакомились с использованием массивов для создания таких типов переменных, в которых содержится несколько объектов или значений. У массивов, однако, имеются определенные ограничения. Наиболее существенным из них является то, что после создания массива его размеры должны оставаться неизменными, поэтому дописывать новые элементы в конец уже существующего массива нельзя — требуется создавать новый массив. В результате синтаксис, используемый для работы с массивами, часто оказывается чрезмерно перегруженным. Методы ООП позволяют создавать классы, которые выполняют существенную часть этих манипуляций внутренним образом, упрощая тем самым код, в котором используются списки элементов или массивы.

Массивы в C# реализованы как экземпляры класса `System.Array` и представляют собой просто один из тех типов, которые известны под названием **классов семейств**. Классы семейств, вообще говоря, используются для работы со списками объектов и по сравнению с обычными массивами обладают некоторыми дополнительными функциональными возможностями. Эти возможности возникают за счет реализации интерфейсов из пространства имен `System.Collections`, что позволяет стандартизировать синтаксис семейств. В пространстве имен содержатся также и некоторые другие интересные вещи, например, классы, которые реализуют интерфейсы отличными от `System.Array` способами.

Функциональные возможности семейств, в том числе такие основополагающие функции, как доступ к членам семейств с использованием синтаксиса `[`*индекс*`]`, реализуются посредством интерфейсов, причем мы не ограничены базовыми классами семейств, таких как `System.Array`. Напротив, у пользователей имеется возможность создавать свои собственные классы семейств. Такие классы могут в большей степени соответствовать объектам, с которыми их планируется использовать (т. е. тем объектам, из которых планируется создавать семейство). Одно из преимуществ такого подхода заключается в том, что, как мы увидим в дальнейшем, создаваемые пользователем классы семейств могут быть **строго типизированы**. Это означает, что при извлечении элементов из семейства не требуется приводить их типы к требуемому типу.

В пространстве имен `System.Collections` существуют интерфейсы, которые обеспечивают выполнение базовых функциональных возможностей семейств:

- `IEnumerable` позволяет просматривать элементы семейств в цикле.
- `ICollection` (наследуемый от `IEnumerable`) позволяет получать общее число элементов в семействе, а также копировать элементы семейства в обычный массив.
- `IList` (который наследуется от `IEnumerable` и `ICollection`) предоставляет список элементов семейства и доступ к этим элементам, а также некоторые другие базовые возможности, имеющие отношение к работе со списками элементов.
- `IDictionary` (наследуемый от `IEnumerable` и `ICollection`) аналогичен интерфейсу `IList`, но предоставляет список элементов, доступ к которым осуществляется не с помощью индекса, а посредством некоторого ключевого значения.

Класс `System.Array` наследуется от `IList`, `ICollection` и `IEnumerable`, однако он не поддерживает некоторые наиболее мощные возможности интерфейса `IList` и представляет список элементов только фиксированной длины.

Использование семейств

Класс `System.Collections.ArrayList` — один из классов в пространстве имен `System.Collections` — тоже реализует интерфейсы `IEnumerable`, `ICollection` и `IList`, однако делает это более изощренным способом, чем класс `System.Array`. В то время как массивы обладают фиксированным размером (мы не имеем возможности добавлять или исключать элементы), этот класс может использоваться для представления списков элементов переменной длины. Чтобы получить представление о том, какими возможностями обладают такие усложненные семейства, давайте рассмотрим пример, в котором наряду с таким классом будет использоваться и обыкновенный массив.

Практикум: сравнение массивов и более сложных семейств

1. Создайте новое консольное приложенне с именем `Ch11Ex01` в директории `C:\BegCSharp\Chapter11`.

2. Добавьте иовый класс `Animal` (животное) к проекту в файле `Animal.cs`, воспользовавшись мастером Add Class.

3. Внесите следующие изменения в `Animal.cs`:

```
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";
      }
      public Animal(string newName)
      {
         name = newName;
      }

      public void Feed()
      {
         Console.WriteLine("{0} has been fed.", name);
      }
   }
   public class Cow : Animal
   {
      public void Milk()
      {
         Console.WriteLine("{0} has been milked.", name);
      }

      public Cow(string newName) : base(newName)
      {
      }
   }

   public class Chicken : Animal
   {
      public void LayEgg()
      {
         Console.WriteLine("{0} has laid an egg.", name);
      }

      public Chicken(string newName) : base(newName)
      {
      }
   }
}
```

4. Внесите следующие изменения в Class1.cs:

```
using System;
using System.Collections;

namespace Ch11Ex01
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Create an Array type collection of Animal " +
            "objects and uee it:");

         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);
         }

         Console.WriteLine("Array collection contains {0} objects.",
            animalArray.Length);
         animalArray[0].Feed();
         ((Chicken)animalArray[1]).LayEgg();
         Console.WriteLine();

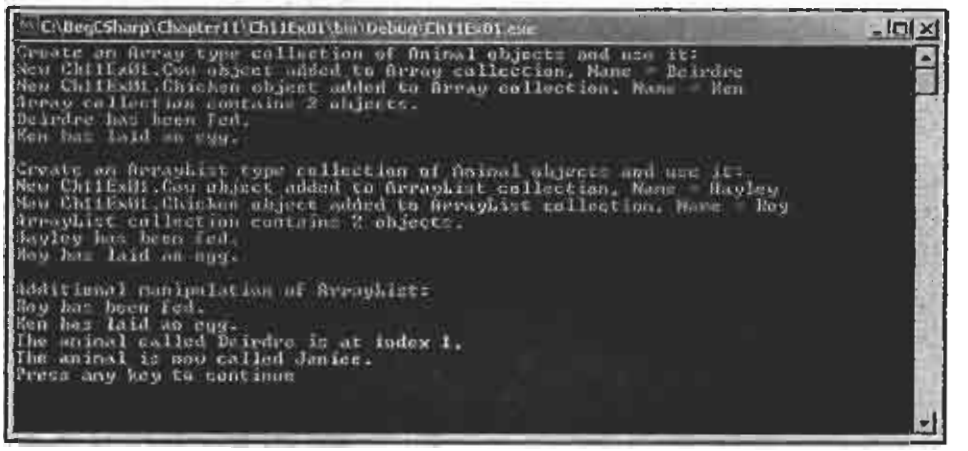
         Console.WriteLine("Create an ArrayList type collection of Animal " +
            "objects and use it:");
         ArrayList animalArrayList = new ArrayList();
         Cow myCow2 = new Cow("Hayley");
         animalArrayList.Add(myCow2);
         animalArrayList.Add(new Chicken("Roy"));

         foreach (Animal myAnimal in animalArrayList)
         {
            Console.WriteLine("New {0} object added to ArrayList collection," +
               " Name = {1}", myAnimal.ToString(), myAnimal.Name);
         }

         Console.WriteLine("ArrayList collection contains {0} objects.",
            animalArrayList.Count);
         ((Animal)animalArrayList[0]).Feed();
         ((Chicken)animalArrayList[1]).LayEgg();
         Console.WriteLine();

         Console.WriteLine("Additional manipulation of 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));
         myCow1.Name = "Janice";
         Console.WriteLine("The animal is now called {0}.",
            ((Animal)animalArrayList[1]).Name);
      }
   }
}
```

5. Запустите приложение:



Как это работает

В этом примере создается два семейства объектов, причем первое — с использованием класса `System.Array` (т. е. обыкновенный массив), а второе — с использованием класса `System.ArrayList`. Оба они являются семействами объектов класса `Animal`, который определен в файле `Animal.cs`. Класс `Animal` является абстрактным классом, следовательно, создание экземпляров этого класса является недопустимым; однако за счет использования полиморфизма имеется возможность включить в состав семейства экземпляры класса `Cow` и класса `Chicken`, которые являются производными от класса `Animal`.

Эти массивы создаются в методе `Main()` в файле `Class1.cs`, после чего над ними производятся различные манипуляции, показывающие их характеристики и возможности. Некоторые из продемонстрированных операций применимы как к семейству `Array`, так и к семейству `ArrayList`, хотя и имеются незначительные отличия в синтаксисе. Однако есть и такие операции, выполнение которых оказывается возможным только с более сложным типом `ArrayList`.

Мы начнем с рассмотрения простых операций, сравнивая код и полученные результаты для обоих типов семейств.

Итак, первое — создание семейства. Для того чтобы использовать обычный массив, необходимо инициализировать его с фиксированными размерами. Мы выполняем эти действия над массивом с именем `animalArray`, применяя стандартный синтаксис из главы 5:

```
Animal[] animalArray = new Animal[2];
```

С другой стороны, для семейств `ArrayList` при инициализации задание размера не требуется, поэтому список (с именем `animalArrayList`) может быть создан следующим образом:

```
ArrayList animalArrayList = new ArrayList();
```

Существует два других конструктора, которые могут быть использованы для данного класса. Первый осуществляет копирование содержимого уже существующего семейства в новый экземпляр, при этом семейство передается ему в качестве параметра; второй конструктор позволяет задать **емкость** семейства и тоже посредством параметра. Эта емкость, задаваемая значением типа `int`, устанавливает начальное количество элементов, которые могут содержаться в данном семействе.

На самом деле это не является реальной емкостью семейства, поскольку емкость автоматически удваивается, если число элементов превышает заданное значение.

В случае массивов, состоящих из элементов ссылочных типов (как используемые объекты класса `Animal` и производных от него классов), простая инициализация массива с указанием размера не влечет за собой инициалнзации его элементов. Для того чтобы воспользоваться некоторым вхождением, его также необходимо инициализировать, откуда следует, что элементам массива должны быть присвоеиы инициализированные объекты:

```
Cow myCow1 = new Cow("Deirdre");
animalArray[0] = myCow1;
animalArray[1] = new Chicken("Ken");
```

В этом коде процедура выполняется двумя способами: один раз — путем присваивання уже существующего объекта класса `Cow` и второй раз — через создание нового объекта класса `Chicken`. Осиовное отличие между этими способами заключается в том, что в первом случае мы получаем ссылку на объект, находящийся в массиве; этот факт мы используем в нашей программе позднее.

В семействе `ArrayList` не существует никаких элементов, даже ссылающихся на `null`. Это означает, что мы лишены возможности присваивать новые объекты аналогичным образом, с использованием индексов. Вместо этого для включения новых объектов необходимо воспользоваться методом `Add()` объекта `ArrayList`:

```
Cow myCow2 = new Cow("Hayley");
animalArrayList.Add(myCow2);
animalArrayList.Add(new Chicken("Roy"));
```

Несколько изменив синтаксис, можно точно таким же образом добавлять в семейство новые или уже существующие объекты.

После того как элементы добавлены, появляется возможность перезаписать их, используя синтаксис, идентичный применяемому для работы с массивами. Например:

```
animalArrayList[0] = new Cow("Alma");
```

Правда, в настоящем примере мы таких действий предпринимать не будем.

В главе 5 вы познакомились с тем, как использовать конструкцию `foreach` для прохода по элементам массива. Это оказывается возможным, поскольку класс `System.Array` реализует интерфейс `IEnumerable`, а единственный метод этого интерфейса, `GetEnumerator()`, позволяет проходить в цикле по всем элементам семейства. Более подробно мы будем рассматривать этот вопрос позже. В нашей программе мы выводим информацию обо всех объектах класса `Animal`, содержащихся 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, следовательно, может использоваться вместе с конструкцией `foreach`:

```
foreach (Animal myAnimal in animalArrayList)
{
   Console.WriteLine("New {0} object added to ArrayList collection," +
      " Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

Далее для вывода на экран числа элементов, содержащихся в массиве, мы используем свойство массива `Length`:

```
Console.WriteLine("Array collection contains {0} objects.",
   animalArray.Length);
```

Мы можем достигнуть того же самого результата и для семейства `ArrayList`, с тем лишь отличием, что придется воспользоваться свойством `Count`, которое является составной частью интерфейса `ICollection`:

```
Console.WriteLine("ArrayList collection contains {0} objects.",
   animalArrayList.Count);
```

Семейства — независимо от того, являются ли они массивами или более сложными семействами — оказываются не столь полезными, если в них не предусмотрено обеспечение доступа к принадлежащим им элементам. Простые массивы строго типизированы, т. е. они обеспечивают непосредственный доступ к типу своих элементов. Это означает, что можно напрямую вызывать метод их элемента:

```
animalArray[0].Feed();
```

Поскольку массив имеет абстрактный тип `Animal`, то нельзя непосредственно вызывать методы, предоставляемые производными классами. Вместо этого следует воспользоваться приведением типа:

```
((Chicken)animalArray[1]).LayEgg();
```

Семейство `ArrayList` — это семейство объектов класса `System.Object` (мы присвоили ему объекты `Animal`, воспользовавшись полиморфизмом). Это значит, что необходимо выполнить приведение типа для всех его элементов:

```
((Animal)animalArrayList[0]).Feed();
((Chicken)animalArrayList[1]).LayEgg();
```

В оставшейся части кода используются некоторые особенности семейства `Animal`, которые отсутствуют у семейства `Array`.

Во-первых, можно удалять элементы с помощью методов `Remove()` и `RemoveAt()`, которые являются составной частью реализации класса `ArrayList`. Эти методы производят удаление элемента из массива по ссылке и по индексу соответственно. В нашем примере для удаления первого включенного в список элемента — объекта класса `Cow`, свойство `Name` которого имеет значение `"Hayley"` — используется второй метод:

```
animalArrayList.RemoveAt(0);
```

В качестве альтернативы можно было бы применить следующий код:

```
animalArrayList.Remove(myCow2);
```

Поскольку на этот объект уже имеется локальная ссылка, то существующую ссылку мы добавили в массив вместо того, чтобы создавать новый объект.

Как бы то ни было, единственным элементом, оставшимся в семействе, является объект класса `Chicken`, доступ к которому мы осуществляем следующим образом:

```
((Animal)animalArrayList[0]).Feed();
```

Выполнение любых операций над элементами объекта `ArrayList`, в результате которых в этом массиве останется `N` элементов, будет производиться таким образом, что этим элементам будут соответствовать индексы в диапазоне от 0 до `N-1`.

Так, например, удаление элемента с номером 0 приводит к тому, что все остальные элементы сдвигаются в массиве на одну позицию, поэтому доступ к объекту класса Chicken осуществляется с индексом 0, а не 1. А поскольку теперь в массиве не существует элемента с индексом 1 (с самого начала было всего два элемента), то при попытке выполнить следующий код будет сгенерирована исключительная ситуация:

```
((Animal)animalArrayList[1]).Feed();
```

Семейства ArrayList позволяют добавлять сразу по несколько элементов с помощью метода AddRange(). Этому методу может передаваться произвольный объект, обладающий интерфейсом ICollection, в который входит массив animalArray:

```
animalArrayList.AddRange(animalArray);
```

Теперь для подтверждения того, что это работает, мы можем попытаться осуществить доступ к третьему элементу семейства, который будет являться вторым элементом в animalArray:

```
((Chicken)animalArrayList[2]).LayEgg();
```

Метод AddRange() не является частью ни одного из интерфейсов, предоставляемых классом ArrayList. Этот метод присущ именно классу ArrayList и демонстрирует тот факт, что в классах семейств мы можем описывать определяемое пользователем поведение, которое далеко выходит за рамки рассмотренных выше интерфейсов. Этот класс предоставляет также множество других интересных интерфейсов, например, InsertRange(), который позволяет вставлять массив объектов в любую точку списка, а также методы для выполнения таких задач, как сортировка и изменение порядка следования элементов.

Наконец, мы вновь задействуем возможность использовать несколько ссылок на один и тот же объект. Применив метод IndexOf(), являющийся частью интерфейса IList, мы можем узнать не только то, что теперь объект myCow1 (который мы изначально добавили к массиву animalArray) входит в семейство animalArrayList, но также и каков его индекс:

```
Console.WriteLine("The animal called (0) is at index (1).",
   myCow1.Name, animalArrayList.IndexOf(myCow1));
```

Следующие две строки кода переименовывают объект с помощью ссылки на него и выводят новое имя на экран, используя для этого ссылку семейства:

```
myCow1.Name = "Janice";
Console.WriteLine("The animal is now called (0).",
   ((Animal)animalArrayList[1]).Name);
```

Определение семейств

Теперь, когда мы познакомились с теми возможностями, которые предоставляются при использовании более сложных классов семейств, настала пора узнать, каким образом можно создавать собственные строго типизированные семейства. Один из способов — это реализовать все необходимые методы вручную, однако такой подход может потребовать очень больших затрат времени, а в некоторых случаях оказаться чрезвычайно сложным. В качестве альтернативы существует возможность создать семейство, являющееся производным от некоторого класса, например, от System.Collections.CollectionBase — абстрактного класса, в котором реализована большая часть семейств. Мы рекомендуем использовать именно такой способ.

Класс `CollectionBase` предоставляет интерфейсы `IEnumerable`, `ICollection` и `IList`, однако в них реализованы только некоторые из необходимых методов, а именно методы `Clear()` и `RemoveAt()` интерфейса `IList`, а также свойство `Count` интерфейса `ICollection`. Если возникает необходимость в каких-либо дополнительных функциональных возможностях, то их приходится реализовывать самостоятельно.

Для этого в классе `CollectionBase` предусмотрено два защищенных свойства, которые позволяют осуществлять доступ собственно к хранящимся объектам. Можно воспользоваться свойством `List` — оно обеспечивает доступ к элементам посредством интерфейса `IList`, и свойством `InnerList` — оно является объектом `ArrayList` и используется для хранения элементов.

Например, класс семейства для хранения объектов `Animal` может быть в общих чертах определен следующим образом (вскоре мы познакомимся с более полным определением):

```
public class Animals : CollectionBase
{
   public void Add(Animal newAnimal)
   {
      List.Add(newAnimal);
   }

   public void Remove(Animal oldAnimal)
   {
      List.Remove(oldAnimal);
   }

   public Animals()
   {
   }
}
```

В данном случае `Add()` и `Remove()` реализованы как строго типизированные методы, которые используют стандартный метод `Add()` интерфейса `IList` для осуществления доступа к элементам. Представленные методы будут работать только с классами `Animal` или с классами, производными от `Animal`, в отличие от встречавшихся нам ранее реализаций `ArrayList`, которые допускали использование произвольных объектов.

Класс `CollectionBase` позволяет использовать синтаксис `foreach` для производных семейств. Можно, например, написать следующий код:

```
Console.WriteLine("Using custom collection class 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);
}
```

А вот этот код является недопустимым:

```
animalCollection[0].Feed();
```

Для того чтобы осуществлять доступ к элементам по их индексам таким способом, нам придется воспользоваться **индексатором** (indexer).

Индексаторы

Индексатор — это особый вид свойства, которое можно добавлять в класс для обеспечения способа доступа, аналогичного доступу к массивам. В действительности индексаторы позволяют реализовать еще более сложные способы доступа, поскольку при необходимости разрешают определять и применять сложные типы параметров с использованием квадратных скобок; несмотря на это, применение простого численного индекса для доступа к элементам является наиболее распространенным.

Мы можем добавить индексатор в семейство `Animals` объектов `Animal` следующим образом:

```
public class Animals : CollectionBase
{
   . . .
   public Animal this[int animalIndex]
   {
      get
      {
         return (Animal)List[animalIndex];
      }
      set
      {
         List[animalIndex] = value;
      }
   }
}
```

Ключевое слово `this` используется с параметрами, задаваемыми в квадратных скобках, однако во всем остальном оно выглядит почти точно так же, как и любое другое свойство. Такой синтаксис представляется весьма логичным, поскольку мы собираемся осуществлять доступ к индексатору, используя имя объекта, за которым следует один или несколько параметров, заключенных в квадратные скобки (например, `MyAnimals[0]`).

Этот код использует индексатор для свойства `List` (т. е. для интерфейса `IList`, обеспечивающего доступ к семейству `ArrayList` класса `CollectionBase`, в котором хранятся наши элементы):

```
return (Animal)List[animalIndex];
```

Явное приведение типа является в данном случае совершенно необходимым, поскольку свойство `IList.List` возвращает объект класса `System.Object`.

Следует обратить внимание на то, что нам приходится определять тип индексатора. Это тот тип, который будет получаться при обращении к элементу посредством данного индексатора. Это означает, что можно использовать следующий код:

```
animalCollection[0].Feed();
```

вместо:

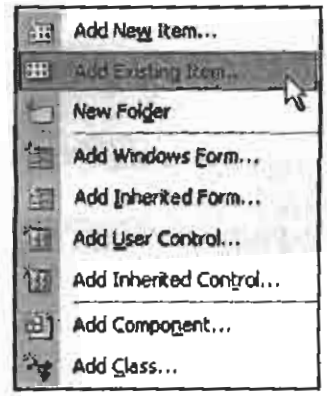
```
((Animal)animalCollection[0]).Feed();
```

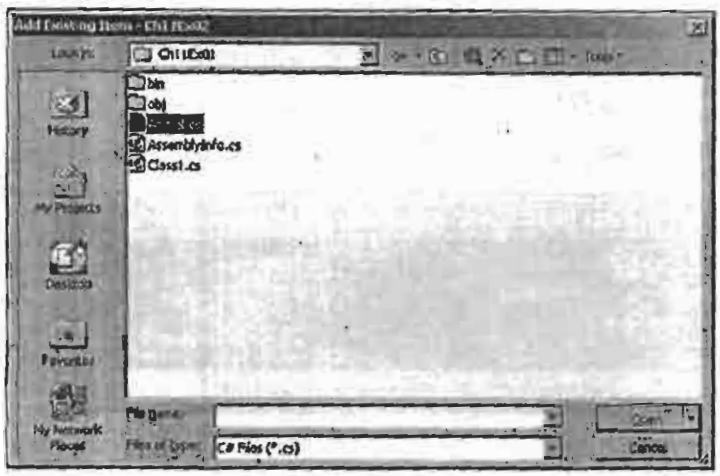
Это еще одна удобная возможность, которую предоставляют строго типизированные семейства, создаваемые пользователем. Давайте расширим наш последний пример, чтобы воспользоваться этим подходом на практике.

Практикум: реализация семейства Animals

1. Создайте новое консольное приложение с именем `Ch11Ex02` в директорин `C:\BegCSharp\Chapter11`.

2. Щелкните правой кнопкой мыши по имени проекта в окне Solution Explorer и выберите пункт Add | Add Existing Item... раскрывающегося меню (см. рис. справа).





3. Выберите файл `Animal.cs` в директории `C:\BegCSharp\Chapter11\Ch11Ex01` и нажмите Open (см. рис. слева).

4. Измените объявление пространства имен в `Animal.cs` следующим образом:

```
namespace Ch11Ex02
```

5. С помощью мастера Add Class добавьте новый класс с именем `Animals` и сохраните его в файле `Animals.cs`.

6. Измените код в `Animals.cs` следующим образом:

```
using System;
using System.Collections;

namespace Ch11Ex02
{
   public class Animals : CollectionBase
   {
      public void Add(Animal newAnimal)
      {
         List.Add(newAnimal);
      }

      public void Remove(Animal newAnimal)
      {
         List.Remove(newAnimal);
      }

      public Animals()
      {
      }

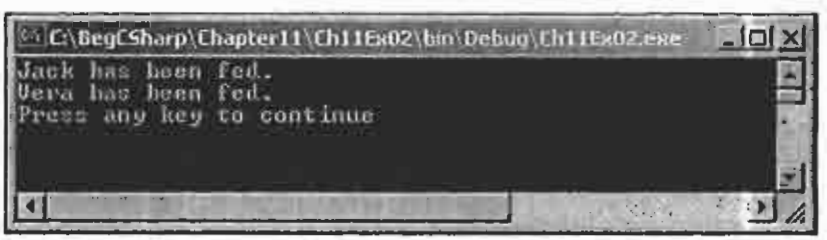
      public Animal this[int animalIndex]
      {
         get
         {
            return (Animal)List[animalIndex];
         }
         set
         {
            List[animalIndex] = value;
         }
      }
   }
}
```

7. Модифицируйте код в Class1.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();
   }
}
```

8. Запустите приложение:

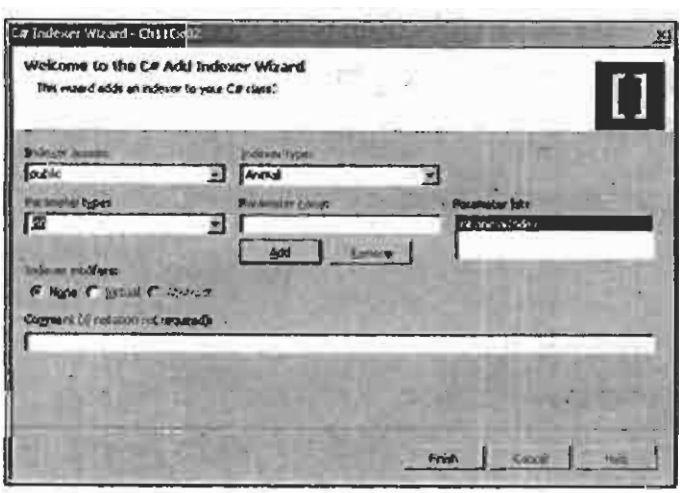


Как это работает

В этом примере используется подробно разобранный в предыдущем разделе код, предназначенный для реализации строго типизированного семейства объектов Animal в классе с именем Animals. Код в Main() создает экземпляр объекта класса Animals с именем animalCollection, добавляет в него два элемента (по одному элементу класса Cow и класса Chicken) и использует цикл foreach для вызова метода Feed(), который оба этих объекта наследуют от своего общего базового класса Animal.

Мастер Add Indexer (добавление индексатора)

В VS существует еще один мастер, используемый в рамках ООП на C#,— Add Indexer. Он располагается в том же месте, где и мастера для добавления свойств, методов и полей (контекстное меню в окне Class View, пункт Add | Add Indexer...). Мастер открывает диалоговое окно, показанное на рисунке слева.



Мастер Add Indexer позволяет задавать уровень защищенности (в данном случае public, поскольку необходимо, чтобы индексатор был доступен из внешнего по отношению к данному семейству кода) и тип индексатора (в данном случае Animal, поскольку требуется, чтобы индексатор был строго типизирован и возвращал элементы класса Animal). Можно также задавать типы и имена параметров; в настоящем примере мы использовали один параметр типа int с именем animalIndex. Наконец, появилась возможность задать какие-либо описатели для данного индексатора: мы можем определить данный индексатор

либо как виртуальный, либо (в случае абстрактного класса) как абстрактный. Если индексатор объявлен как `virtual`, это означает, что переопределение данного индексатора допускается во всех классах, производных от индексируемого класса,— точно так же, как и переопределение виртуальных методов, с которыми вы познакомились в предыдущей главе. Абстрактные индексаторы ничем не отличаются от виртуальных индексаторов, однако для их реализации не задается никакого кода — это обязанность возлагается на классы, являющиеся производными от индексируемого класса.

Изображенные выше установки позволяют получить следующую структуру для индексатора класса `Animals`:

```
public Animal this(int animalIndex)
{
   get
   {
      return null;
   }
   set
   {
   }
}
```

Заметьте, что наполнение блоков `get` и `set` необходимо задавать вручную.

Семейства с доступом по ключу и интерфейс IDictionary

Кроме интерфейса `IList`, в семействах можно реализовать аналогичный интерфейс `IDictionary`, который позволяет осуществлять доступ к элементам по значению ключа (например, по строке с именем) вместо доступа по индексу.

Это тоже достигается с помощью индексатора, однако в данном случае параметр индексатора привязывается к хранящемуся элементу, а не к индексу типа `int`, что позволяет сделать семейство несколько более дружественным по отношению к пользователю.

Точно так же, как и для индексируемых семейств, в данном случае имеется базовый класс, который можно использовать для упрощения реализации интерфейса `IDictionary`,— класс `DictionaryBase`. В этом классе также реализованы интерфейсы `IEnumerable` и `ICollection`, что обеспечивает единые для всех семейств основные возможности манипуляций.

Класс `DictionaryBase`, как и класс `CollectionBase`, реализует некоторые (но не все) члены, полученные посредством поддерживаемых им интерфейсов. Так же, как и в классе `CollectionBase`, в нем реализованы методы `Clear()` и `Count()`, а вот метод `RemoveAt()` — нет. Эта ситуация объясняется тем, что `RemoveAt()` — это метод интерфейса `IList`, и он реализуется для интерфейса `IDictionary`. Однако `IDictionary` обладает методом `Remove()`, являющимся одним из тех методов, которые необходимо реализовать для создаваемых пользователем классов семейств, основанных на классе `DictionaryBase`.

Следующий код являет собой пример альтернативной версии класса `Animals` из предыдущего раздела, который на этот раз в качестве базового класса использует класс `DictionaryBase`. В него входят методы `Add()`, `Remove()` и индексатор доступа 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;
            }
        }
    }
```

Вот какие различия имеются между этими членами:

- `Add()` принимает два параметра: ключ и значение, которые будут храниться совместно. У данного семейства, построенного по алфавитному принципу, есть член с именем `Dictionary`, наследуемый от класса `DictionaryBase`. Он является интерфейсом `IDictionary`, имеющим собственный метод `Add()` с двумя параметрами типа `object`. Наша реализация предусматривает передачу значения типа `string` в качестве ключа и передачу объекта типа `Animal`.
- `Remove()` получает в качестве параметра ключ, а не ссылку на объект. Элемент, обладающий значением, заданным с помощью ключа, удаляется.
- Индексатор использует вместо индекса значение ключа в виде строки, что позволяет осуществлять доступ к хранящемуся элементу через наследуемый член `Dictionary`. Повторим еще раз, что в данном случае приведение типа является обязательным.

Еще одним отличием семейств с базовым классом `DictionaryBase` от семейств с базовым классом `CollectionBase` является то, что для них немного по-разному выполняется конструкция `foreach`. Семейство из предыдущего раздела допускает непосредственное извлечение объектов типа `Animal`. Использование конструкции `foreach` с классом, производным от `DictionaryBase`, приводит к получению структур типа `DictionaryEntry` — еще одного типа, описанного в пространстве имен `System.Collections`. Для того чтобы добраться до самих объектов типа `Animal`, необходимо воспользоваться членом этой структуры `Value`, хотя для получения доступа к соответствующему ключу можно также использовать член данной структуры `Key`. Теперь использовавшемуся ранее коду:

```
foreach (Animal myAnimal in animalCollection)
{
    Console.WriteLine("New {0} object added to custom collection, " +
        "Name = {1}", myAnimal.ToString(), myAnimal.Name());
}
```

будет эквивалентен следующий код:

```
foreach (DictionaryEntry myEntry in animalCollection)
{
   Console.WriteLine("New {0} object added to custom collection, " +
      "Name = {1}", myEntry.Value.ToString(),
      ((Animal)myEntry.Value).Name);
}
```

На самом деле можно переопределить это поведение таким образом, чтобы извлекать объекты непосредственно с помощью конструкции `foreach`, однако данная тема чрезвычайно сложна, и мы ее рассматривать не будем.

Посвященная этому вопросу статья Карли Ватсона опубликована на web-сайте `CSharpToday.com`.

Усовершенствование CardLib, часть 1

В предыдущей главе был создан проект библиотеки классов под названием `Ch10CardLib`, в котором содержался класс `Card`, представляющий игральные карты, и класс `Deck`, представляющий колоду карт, т. е. являющийся семейством объектов класса `Card`. Это семейство было реализовано в виде простого массива.

Теперь добавим в эту библиотеку, которую переименуем в `Ch11CardLib`, еще один класс. Этот новый класс — назовем его `Cards` — будет представлять созданное пользователем семейство объектов типа `Card`, что позволит использовать преимущества, о которых рассказывалось в данной главе. Код, описывающий этот класс, находится в файле `Cards.cs` и имеет следующий вид (изменения в коде, созданном мастером, выделены серым цветом):

```
using System;
using System.Collections;

namespace Ch11CardLib
{
   /// <summary>
   /// Summary description for Cards.
   /// </summary>
   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;
         }
      }
```

```
            // Проверка, позволяющая определять, содержится ли в семействе
            // Cards конкретная карта. С этой целью осуществляется вызов метода
            // Contains класса ArrayList для данного семейства, доступ к
            // которому осуществляется посредством InnerList.
            public bool Contains(Card card)
            {
               return InnerList.Contains(card);
            }
         }
      }
```

Теперь нам предстоит внести определенные изменения в файл Deck.cs, чтобы он мог использовать это новое семейство вместо массива:

```
using System;

namespace Ch11CardLib
{
   public class Deck
   {
      private Cards cards = new Cards();

      public Deck()
      {
         // здесь располагалась удаленная 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,
               "Значение должно находиться в диапазоне между 0 и 51."));
      }

      public void Shuffle()
      {
         Cards newDeck = new Cards();
         bool[] assigned = new bool[52];
         for (int i = 0; i < 52; i++)
         {
            int sourceCard = 0;
            bool foundCard = false;
            Random sourceGen = new Random();
            while (foundCard == false)
            {
               sourceCard = sourceGen.Next(52);
               if (assigned[sourceCard] == false)
                  foundCard = true;
            }
            assigned[sourceCard] = true;
            newDeck.Add(cards[sourceCard]);
         }
         cards = newDeck;
      }
   }
}
```

Обратите внимание, что при копировании исходных файлов из `Ch10CardLib` в `Ch11CardLib` необходимо изменить объявления пространства имен таким образом, чтобы они ссылались на `Ch11CardLib`; именно поэтому строка с объявлением пространства имен в коде также выделена. Это справедливо как для файла `Card.cs`, так и для консольного приложения `Ch10CardClient`.

В данном случае потребовалось внести не так уж много изменений. Большая их часть касается изменения логики процедуры тасования колоды. Теперь карты выбираются по случайному индексу и добавляются в начало нового семейства элементов `Cards` с именем `newDeck`, а раньше они выбирались по последовательным индексам из `cards` и добавлялись в `newDeck` по индексу, определяемому случайным образом.

Клиентское консольное приложение `Ch10CardClient` для решения `Ch10CardLib` может работать с новой библиотекой; при этом результаты окажутся точно такими же, поскольку сигнатуры методов в `Deck` остались неизменными. Клиенты этой библиотеки классов теперь могут использовать класс семейства `Cards`, вместо того чтобы работать с массивами объектов типа `Card`, например, при определении розданных игрокам карт в приложении для карточных игр.

Перегрузка операторов

Следующим предметом рассмотрения будет **перегрузка операторов**. Она позволяет применять стандартные операторы — такие как +, > и т. п.— в операциях над созданными нами классами. Это называется перегрузкой, поскольку мы используем нашу собственную реализацию работы этих операторов в тех случаях, когда они применяются для выполнения операций над параметрами определенных типов; во многом это аналогично тому, как перегружаются методы, когда передаются различные параметры методам с одним и тем же именем.

Перегрузка операторов оказывается весьма полезной, поскольку позволяет выполнять любые действия, которые мы определим при реализации перегрузки оператора. Они могут быть вовсе не так просты, как, скажем, "+ означает сложение двух данных операндов". Немного позже, при дальнейшем усовершенствовании библиотеки `CardLib`, нам встретится хороший пример такого подхода. Мы включим в нее такие реализации операторов сравнения, которые позволят сравнивать две карты, чтобы определить, какая из них бьет другую и берет взятку (этап карточной игры). Поскольку взятка во многих карточных играх зависит от мастей участвующих в ней карт, то эта задача оказывается не такой примитивной, чтобы просто сравнивать старшинство карт. Если вторая сыгранная карта отличается по масти от первой, то первая карта забирает взятку независимо от старшинства. Такая логика может быть реализована, если учитывать порядок следования двух операндов. Мы можем также ввести понятие козырной масти: козыри бьют все остальные масти даже в том случае, если это не первая сыгранная карта. Это означает, что если результат `card1 > card2` равен `true` (т. е. `card1` бьет `card2`, если `card1` сыграна первой), из него совершенно не следует, что значение `card2 > card1` есть `false`. Если ни `card1`, ни `card2` не являются козырями и масти их различны, то оба эти сравнения дадут значение `true`.

Однако для начала давайте познакомимся с основным синтаксисом перегрузки операторов.

Операторы могут перегружаться с помощью включения в класс членов типа оператора (которые должны быть описаны как `static`). Для некоторых операторов возможны различные варианты использования (например, для оператора — он может использоваться и как унарный, и как бинарный), поэтому необходимо указывать, с каким количеством операндов мы имеем дело и какого они типа. Обычно работа ведется с операндами того же типа, что и класс, в котором определяется данный оператор, хотя возможна работа со смешанными типами, с чем мы познакомимся позже.

В качестве примера давайте рассмотрим простой класс с именем `AddClass1`, определенный следующим образом:

```
public class AddClass1
{
   public int val;
}
```

На самом деле это просто оболочка для значения типа `int`, однако она вполне может служить для иллюстрации общих принципов.

Для данного класса код следующего вида не сможет пройти компиляцию:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
AddClass1 op3 = op1 + op2;
```

В полученном сообщении об ошибке будет содержаться информация о том, что оператор + не может использоваться с операндами типа `AddClass1`, поскольку мы не определили выполняемую операцию.

Следующий код будет работать, хотя он может приводнть к получению совсем не того результата, который вы ожидаете:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
bool op3 = op1 == op2;
```

В данном случае `op1` и `op2` сравниваются с помощью бинарного оператора `==`, который позволяет определить, ссылаются ли они на один и тот же объект, а вовсе не то, обладают ли они одинаковыми значениями. `op3` будет присвоено в этом примере значение `false`, несмотря на то, что `op1.val` и `op2.val` имеют идентичные значения.

Чтобы перегрузить оператор +, нам потребуется создать следующий 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 легко можно заметить, перегрузка оператора очень напоминает объявление стандартного статичного метода, за исключением того, что в данном случае используется ключевое слово `operator` и сам оператор вместо имени метода.

Теперь мы можем с успехом применить оператор + с операндами данного класса, что мы пытались сделать в предыдущем примере:

```
AddClass1 op3 = op1 + op2;
```

Перегрузка всех остальных бинарных операторов осуществляется по той же схеме. Перегрузка унарных операторов аналогична, только в этом случае используется единственный 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;
   }
}
```

Оба этих оператора работают с операндами того же типа, что и сам класс, и возвращают значения этого же типа. Давайте, однако, рассмотрим определения следующих классов:

```
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;
}
```

В этом случае следующий код будет вполне допустимым:

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass2 op2 = new AddClass2();
op2.val = 5;
AddClass3 op3 = op1 + op2;
```

Там, где это необходимо, имеется возможность использовать таким образом различные типы. Обратите, однако, внимание, что если мы добавим тот же оператор в класс `AddClass2`, то приведенный выше код работать не будет, поскольку

невозможно будет понять, какой именно из операторов следует использовать. Отсюда следует, что мы обязаны следить за тем, чтобы операторы с одинаковой сигнатурой были описаны не более чем в одном классе.

Необходимо следить и за тем, чтобы в случае использования операндов различных типов операнды следовали в том же порядке, что и параметры при описании оператора. Если мы попытаемся воспользоваться перегруженным оператором, задав операнды в "неправильном" порядке, операция выполнена не будет. Следующий оператор является недопустимым:

```
AddClass3 op3 = op2 + op1;
```

если, конечно, мы не включим еще одну перегрузку с измененным порядком следования параметров:

```
public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)
{
   AddClass3 returnVal = new AddClass3();
   returnVal.val = op1.val + op2.val;
   return returnVal;
}
```

Перегрузка допускается для следующих операторов:

- Унарные операторы: +, -, !, ~, ++, `true`, `false`
- Бинарные операторы: +, -, *, /, %, &, |, ^, <<, >>
- Операторы сравнения: ==, !=, <, >, <=, >=

Если перегружаются операторы `true` и `false`, то мы получаем возможность использовать классы в логических выражениях, например, `if (op1) {}`.

Не предусмотрена возможность перегружать операторы присваивания, такие как +=, однако эти операторы состоят из простых операторов (в данном случае +), поэтому нам не приходится об этом беспокоиться. Перегрузка оператора + приведет к тому, что оператор += будет работать так, как нам нужно. В него входит также и оператор =, однако перегружать его не имеет никакого смысла, поскольку его использование универсально. Этот оператор, однако, может иметь отношение к определяемым пользователем операциям отношения, о которых мы будем говорить в следующем разделе.

Также не существует возможности перегружать операторы && и ||, однако в этом случае мы снова обращаемся к использованию составляющих их более простых операторов — & и |, и их перегрузки оказывается вполне достаточно.

Некоторые операторы, например, такие как < и >, должны перегружаться парами. Другими словами, мы не можем перегрузить <, если не будет перегружен и >. В большинстве случаев существует возможность просто вызывать данные операторы при описании других, что позволяет уменьшить необходимое число строк кода (и возможных ошибок), например:

```
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.val >= op2.val);
    }

    // Необходимо также реализовать операторы <= и >
}
```

В случае более сложных определений операторов это означает, что мы можем сэкономить определенное число строк кода, но, кроме того, это означает и необходимость вносить какие-либо изменения только в одну реализацию: вторая также будет изменена.

Все вышесказанное относится и к операторам `==` и `!=`, однако для них часто имеет смысл переопределение функций `Object.Equals()` и `Object.GetHashCode()`, поскольку обе они могут использоваться для сравнения объектов. Переопределяя эти методы, мы получаем возможность гарантировать, что независимо от методов, применяемых пользователями данного класса, полученные результаты будут одинаковыми. Это не является необходимым, но это стоит сделать ради завершенности. Для этого потребуются следующие нестатические переопределенные методы:

```
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;
    }
}
```

Обратите внимание, что функции `Equals()` передается параметр типа `object`. Такая сигнатура оказывается необходимой, иначе получится, что мы перегружаем этот метод, вместо того чтобы переопределять его, и реализация по умолчанию по-прежнему будет доступна пользователям данного класса. Это означает, что для получения необходимого результата нам потребуется приведение типов (хотя для придания коду большей стройности могут понадобиться дополнительные действия, поскольку приведенный выше пример не будет работать в тех случаях, когда параметр `op1` не является экземпляром класса `AddClass1` или экземпляром производного от него класса).

Функция `GetHashCode()` применяется для получения значения типа `int`, уникального для данного экземпляра объекта и зависящего от его состояния. В данном случае использование поля `val` оказывается вполне допустимым, поскольку оно также имеет значение типа `int`.

Операторы преобразования типов

Помимо перегрузки математических операторов, продемонстрированной выше, существует возможность определять как неявное, так и явное преобразование типов. Это оказывается необходимым в том случае, если мы хотим выполнять преобразования между типами, не имеющими между собой отношений, например, когда они не связаны иерархией наследования и не имеют общих интерфейсов.

Пусть мы описали неявное преобразование между классом `convclass1` и классом `convclass2`. Это означает, что теперь можно написать следующий код:

```
convclass1 op1 = new convclass1();
convclass2 op2 = op1;
```

В качестве альтернативы мы можем описать явное преобразование, которое будем вызывать с помощью следующего кода:

```
convclass1 op1 = new convclass1();
convclass2 op2 = (convclass2)op1;
```

Давайте в качестве примера 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;
   }
}
```

В данном случае `convclass1` содержит значение типа `int`, а `convclass2` содержит значение типа `double`. Поскольку значения типа `int` могут неявно преобразовываться в значения типа `double`, то существует возможность описать неявное преобразование между `convclass1` и `convclass2`. Однако поскольку обратное неверно, мы вынуждены описывать оператор преобразования `convclass2` в `convclass1` как явный.

В программе это было задано с помощью ключевых слов `implicit` и `explicit`.

Для описанных выше классов следующий код является вполне допустимым:

```
convclass1 op1 = new convclass1();
op1.val = 3;
convclass2 op2 = op1;
```

Однако преобразование в обратном направлении требует следующего явного изменения типа:

```
convclass2 op1 = new convclass2();
op1.val = 3e15;
convclass1 op2 = (convclass1)op1;
```

Обратите внимание, что поскольку при описании явного преобразования нами было использовано ключевое слово `checked`, в приведенном выше коде мы получим исключительную ситуацию, так как значение свойства `val` у `op1` слишком велико и не может быть присвоено свойству `val` `op2`.

Усовершенствование CardLib, часть 2

Теперь давайте снова модернизируем проект `Ch11CardLib`, добавив в него перегрузку операторов для класса, описывающего карты. Для начала, однако, мы добавим в класс `Card` некоторые дополнительные поля, которые позволят нам описывать козырную масть и использовать туз в качестве самой старшей карты. Мы описываем их как `static`, поскольку, после того как они определены, они могут применятся ко всем объектам типа `Card`:

```
public class Card
{
   // Маркер, свидетельствующий об использовании козырей. Если ему
   // присвоено значение true, то козыри имеют более высокое старшинство,
   // чем карты всех остальных мастей.
   public static bool useTrumps = false;

   // Указывает на то, какая именно масть является козырной в том случае,
   // если useTrumps имеет значение true.
   public static Suit trump = Suit.Club;

   // Маркер, который определяет, являются ли тузы старше королей
   // или младше двоек.
   public static bool isAceHigh = true;
```

После этого имеет смысл добавить в класс `Deck` несколько дополнительных конструкторов, которые позволят инициализировать колоды карт с разными характеристиками:

```
public Deck()
{
   for (int suitVal = 0; suitVal < 4; suitVal++)
   {
      for (int rankVal = 1; rankVal < 14; rankVal++)
      {
         cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
      }
   }
}

// Конструктор не по умолчанию. Позволяет устанавливать наивысшее
// старшинство тузов.
public Deck(bool isAceHigh) : this()
{
   Card.isAceHigh = isAceHigh;
}

// Конструктор не по умолчанию. Позволяет использовать козырную масть.
public Deck(bool useTrumps, Suit trump) : this()
{
   Card.useTrumps = useTrumps;
   Card.trump = trump;
}
```

```
// Конструктор не по умолчанию. Позволяет устанавливать наивысшее
// старшинство тузов и использовать козырную масть.
public Deck(bool isAceHigh, bool useTrumps, Suit trump) : this()
{
   Card.isAceHigh = isAceHigh;
   Card.useTrumps = useTrumps;
   Card.trump = trump;
}
```

Каждый из этих конструкторов определен с использованием синтаксиса `: this()`, с которым вы познакомились в главе 9, поэтому во всех случаях первым будет вызываться конструктор по умолчанию, который будет инициализировать колоду.

Далее мы добавляем перегрузку операторов (и предлагаемые переопределения) в класс `Card`:

```
public Card(Suit newSuit, Rank newRank)
{
   suit = newSuit;
   rank = newRank;
}

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)
         return (card1.rank > card2.rank) ^ (card2.rank == Rank.Ace);
      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)
         return (card1.rank >= card2.rank) ^ (card2.rank == Rank.Ace);
      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);
}
```

По этому коду у нас нет особых замечаний, за исключением, пожалуй, несколько усложненного способа использования логического оператора XOR (^). Он применяется, например, при перегрузке оператора >. Если мы шаг за шагом пройдем этот код, то сможем понять, каким образом он работает и почему в данном случае необходим именно он.

Мы производим сравнение двух карт — `card1` и `card2`, причем предполагается, что карта `card1` была сыграна первой. Как уже обсуждалось ранее, это важно в том случае, если мы используем козырные карты, поскольку козырная карта бьет некозырную, даже когда последняя старше. Естественно, если обе карты оказываются одной масти, то уже не имеет значения, является ли данная масть козырной или нет, поэтому именно эта проверка осуществляется в первую очередь:

```
public static bool operator >(Card card1, Card card2)
{
   if (card1.suit == card2.suit)
   {
```

Если статический маркер `isAceHigh` имеет значение `true`, то в этом случае мы лишены возможности сравнивать старшинство карт непосредственно по их значению в перечислимом типе `Rank`, поскольку старшинство туза в этом типе равно 1 (т. е. ниже, чем старшинство всех остальных карт). Именно здесь мы можем очень эффективно использовать оператор ^. Мы выполняем два сравнения: одно по старшинству и одно на предмет того, не является ли карта тузом. Результаты сравнений могут быть следующими:

- Если карта `card1` старше, чем `card2`, и карта `card2` не является тузом, то `card1` старше, чем `card2`.
- Если карта `card1` старше, чем `card2`, и карта `card2` является тузом, то `card2` старше, чем `card1`.
- Если карта `card1` младше `card2` и не является тузом, то `card2` старше, чем `card1`.
- Если карта `card1` младше `card2` и является тузом, то `card1` старше, чем `card2`.

card1 > card2	card2.rank == Rank.Ace	card1 старше, чем card2?
true	false	true
true	true	false
false	false	false
false	true	true

Все вышесказанное можно представить в виде таблицы, расположенной слева.

Напоминаем вам, что оператор xor работает аналогично тому, как представлено в следующей таблице.

op1	op2	op1 ^ op2
true	false	true
true	true	false
false	false	false
false	true	true

В данном случае логика оказывается идентичной, поэтому использование этого оператора позволяет получить требуемый результат:

```
if (isAceHigh)
   return (card1.rank > card2.rank) ^ (card2.rank == Rank.Ace);
```

Однако в том случае, если isAceHigh имеет значение false, мы вполне можем положиться на старшинство карты, что позволяет упростить выполнение операции:

```
        else
           return (card1.rank > card2.rank);
    }
```

Остальная часть кода относится к тому случаю, когда карты card1 и card2 имеют разные масти. При этом оказывается важным статический маркер useTrumps. Если он имеет значение true и карта card2 — козырной масти, то мы можем совершенно определенно утверждать, что карта card1 не является козырной (так как эти карты разных мастей), а поскольку козыри всегда старше, то карта card2 оказывается более старшей:

```
else
{
   if (useTrumps && (card2.suit == Card.trump))
      return false;
```

Если карта card2 не является козырной (т. е. маркер useTrumps имеет значение false), то в этом случае более старшей оказывается карта card1, поскольку именно она была сыграна первой:

```
      else
         return true;
   }
}
```

Еще один оператор (>=) использует аналогичный код, а все остальное совсем просто, поэтому мы не будем вдаваться в подробности.

Следующий простой клиентский код тестирует эти операторы (для этого код следует поместить в функцию Main() клиентского проекта точно так же, как мы поступали с клиентским кодом в предыдущих примерах с использованием CardLib):

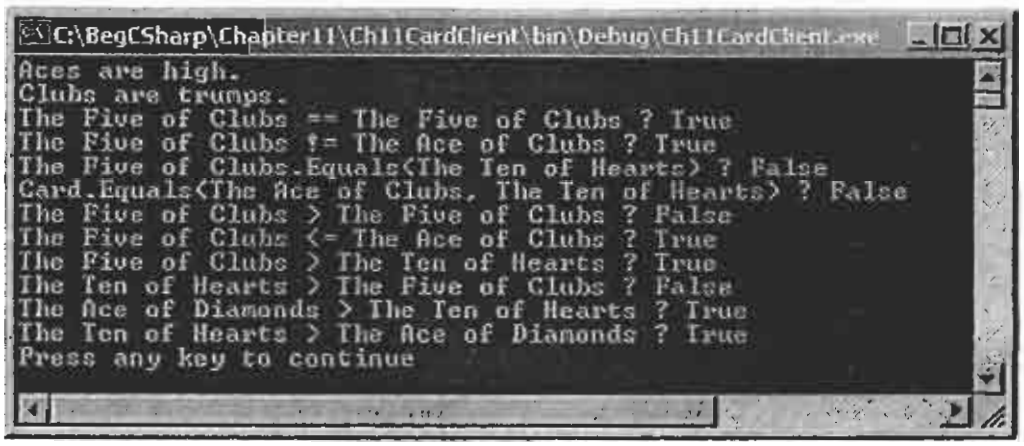
```
Card.isAceHigh = true;
Console.WriteLine("Aces are high.");        // Тузы имеют наивысшее старшинство. //
Card.useTrumps = true;
Card.trump = Suit.Club;
Console.WriteLine("Clubs are trumps.");     // Козырь — трефы. //

Card card1, card2, card3, card4, card5;
card1 = new Card(Suit.Club, Rank.Five);     // пятерка треф //
card2 = new Card(Suit.Club, Rank.Five);     // пятерка треф //
```

```
card3 = new Card(Suit.Club, Rank.Ace);     // туз треф //
card4 = new Card(Suit.Heart, Rank.Ten);    // десятка червей //
card5 = new Card(Suit.Diamond, Rank.Ace);  // туз бубен //

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);
```

Будут получены следующие результаты:



В каждом случае используемые операторы учитывают установленные нами правила. Это очень хорошо видно по последним четырем строкам вывода, в которых демонстрируется, что козырная карта всегда бьет некозырную.

Более сложные преобразования типов

Теперь, когда вы знаете, каким образом описываются операторы преобразования типов, настала пора увенчать ваше знакомство с этой темой рассмотрением некоторых дополнительных возможностей языка C#. В этом разделе мы будем изучать:

- Упаковку и распаковку — преобразования между ссылочными и значимыми типами.

- Оператор `is`, который используется для проверки того, является ли конкретная переменная переменной данного типа или совместимой с данным типом.
- Оператор `as`, который используется для преобразования переменной в указанный тип несколько иным способом, нежели при изменении типа.

Упаковка и распаковка

В главе 8 мы обсуждали отличия между ссылочными и значимыми типами, что было проиллюстрировано в главе 9 путем сравнения структур (которые представляют собой значимый тип) и классов (которые относятся к ссылочным типам). **Упаковка** — это процедура преобразования значимого типа в тип `System.Object` либо в тип интерфейса, который реализован значимым типом. **Распаковка** является обратной процедурой.

Предположим, что имеется структура следующего типа:

```
struct MyStruct
{
   public int Val;
}
```

Мы можем выполнить упаковку этой структуры, поместив ее в переменную типа `object`:

```
MyStruct valType1 = new MyStruct();
valType1.Val = 5;
object refType = valType1;
```

Здесь мы создаем новую переменную (`valType1`) типа `MyStruct`, присваиваем ей значение, а затем осуществляем ее упаковку в переменную типа `object` (`refType`).

В этом объекте, созданном в результате упаковки переменной, содержится ссылка на копию переменной значимого типа, а не ссылка на исходную переменную значимого типа. Это можно проверить, изменив содержимое исходной структуры, выполнив распаковку структуры, которая содержится в переменной типа `object`, и сравнив их значения:

```
valType1.Val = 6;
MyStruct valType2 = (MyStruct)refType;
Console.WriteLine("valType2.Val = {0}", valType2.Val);
```

В результате выполнения этого кода будет получен следующий результат:

```
valType2.Val = 5
```

В том случае, когда объекту присваивается переменная ссылочного типа, то все происходит совсем иначе. Это можно проиллюстрировать, описав `MyStruct` как класс (проигнорировав тот факт, что имя этой переменной оказывается несколько неподходящим для класса):

```
class MyStruct
{
   public int Val;
}
```

Запустив приведенный выше клиентский код без каких бы то ни было изменений (по-прежнему игнорируя некорректно названные переменные), мы получим следующий выходной поток:

```
valType2.Val = 6
```

Также можно осуществлять упаковку переменных в интерфейс, если этот интерфейс в них реализован. Предположим, что тип `MyStruct` реализует интерфейс `IMyInterface` следующим образом:

```
interface IMyInterface
{
}

struct MyStruct : IMyInterface
{
   public int Val;
}
```

В таком случае можно выполнить упаковку этой структуры в тип `IMyInterface`:

```
MyStruct valType1 = new MyStruct();
IMyInterface refType = valType1;
```

и ее распаковку с использованием обычного приведения типа:

```
MyStruct valType2 = (MyStruct)refType;
```

Как можно заметить по этим примерам, процедура упаковки выполняется без вмешательства с нашей стороны (другим словами, отсутствует необходимость писать какой-либо код). А вот процедура распаковки требует выполнения явного преобразования типов, т. е. нужно выполнить приведение типа (упаковка выполняется неявно и таких требований не предъявляет).

Возможно, вы недоумеваете, для чего все это может понадобиться. На самом деле существуют две причины считать процедуру упаковки чрезвычайно полезной. Во-первых, она позволяет использовать значимые типы в семействах (например, в `ArrayList`), в которых собраны элементы ссылочного типа. Во-вторых, она представляет собой некий внутренний механизм, позволяющий вызывать методы типа `object` для значимых типов вроде `int` или `struct`.

И последнее замечание, которое необходимо сделать: выполнение процедуры распаковки является необходимым условием для осуществления доступа к содержимому значимых типов.

Оператор is

Оператор `is` проверяет, можно ли преобразовать какую-либо неизвестную переменную (например, переданную методу в качестве параметра-объекта) в указанный тип; при этом значение `true` свидетельствует, что такое преобразование осуществимо. Этот оператор позволяет, прежде чем вызывать методы на объект, проверять, реализован ли такой метод в типе объекта.

Оператор `is` проверяет *не идентичность* типов, а их совместимость.

Оператор `is` использует следующий синтаксис:

<операнд> `is` *<тип>*

Результаты использования такого выражения могут быть следующие:

- Если *<тип>* — это тип класса, то результат `true` будет получен в том случае, если *<операнд>* имеет тот же самый тип, наследуется от этого типа или допускает упаковку в такой тип.

- Если *<тип>* — это тип интерфейса, то результат `true` будет получен, если *<операнд>* имеет тот же самый тип или тип, в котором данный интерфейс реализован.
- Если *<тип>* — это значимый тип, то результат `true` будет получен, если *<операнд>* имеет тот же самый тип или если его тип допускает распаковку в указанный тип.

Давайте рассмотрим несколько примеров, чтобы увидеть, как все это работает на практике.

Практикум: использование оператора is

1. Создайте новое консольное приложение с именем `Ch11Ex03` в директории `C:\BegCSharp\Chapter11`.
2. Модифицируйте код в `Class1.cs` следующим образом:

```
namespace Ch11Ex03
{
   class Checker
   {
      public void Check(object param1)
      {
         if (param1 is ClassA)
            Console.WriteLine("Variable can be converted to ClassA.");
         else
            Console.WriteLine("Variable can't be converted to ClassA.");
         if (param1 is IMyInterface)
            Console.WriteLine("Variable can be converted to IMyInterface.");
         else
            Console.WriteLine("Variable can't be converted to IMyInterface.");
         if (param1 is MyStruct)
            Console.WriteLine("Variable can be converted to MyStruct.");
         else
            Console.WriteLine("Variable can't be converted to MyStruct.");
      }
   }

   interface IMyInterface
   {
   }

   class ClassA : IMyInterface
   {
   }

   class ClassB : IMyInterface
   {
   }

   class ClassC
   {
   }

   class ClassD : ClassA
   {
   }

   struct MyStruct : IMyInterface
   {
   }
```

```
        /// <summary>
        /// Summary description for Class1.
        /// </summary>
        class Class1
        {
            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:");
                check.Check(try1);

                Console.WriteLine("\nAnalyzing ClaseB type variable:");
                check.Check(try2);
                Console.WriteLine("\nAnalyzing ClassC type variable:");
                check.Check(try3);
                Console.WriteLine("\nAnalyzing ClassD type variable:");
                check.Check(try4);
                Console.WriteLine("\nAnalyzing MyStruct type variable:");
                check.Check(try5);
                Console.WriteLine("\nAnalyzing boxed MyStruct type variable:");
                check.Check(try6);
            }
        }
    }
```



3. Выполните код (см. рис. слева).

Как это работает

Этот пример иллюстрирует различные результаты, к которым может привести применение оператора `is`. В программе определяются четыре класса, интерфейс и структура, в дальнейшем используемые как параметры метода класса, в котором оператор `is` проверяет, могут ли они быть преобразованы в тип `ClassA`, в тип интерфейса и в тип структуры.

Совместимыми с классом `ClassA` являются только сам `ClassA` и `ClassD` (он наследуется от класса `ClassA`). Типы, которые не наследуются от данного класса, не являются совместимыми с названным классом.

В типах `ClassA`, `ClassB` и `MyStruct` реализован интерфейс `IMyInterface`, поэтому все они совместимы с типом `IMyInterface`. Класс `ClassD` наследуется от класса `ClassA`, поэтому он также является совместимым. Следовательно, несовместимым с типом `IMyInterface` оказывается только `ClassC`.

Наконец, только переменные собственно типа `MyStruct` и переменные, полученные в результате упаковки этого типа, оказываются совместимыми с `MyStruct`, поскольку возможности преобразовывать ссылочные типы в значимые типы не существует (за исключением того, что мы можем выполнить процедуру распаковки для тех переменных, для которых ранее была выполнена процедура упаковки).

Оператор as

Оператор `as` осуществляет преобразование типа в заданный ссылочный тип с помощью следующего синтаксиса:

<операнд> `as` *<тип>*

Использование этой возможности допускается только при наличии определенных обстоятельств:

- Если *<операнд>* имеет тип *<тип>*.
- Если *<операнд>* может быть неявно преобразован в тип *<тип>*.
- Если *<операнд>* может быть преобразован в тип *<тип>* в результате процедуры упаковки.

Если *<операнд>* может быть преобразован в *<тип>* явным образом, то результатом вычисления этого выражения будет `null`. Например, в предыдущей программе существует возможность явного преобразования между типами `ClassA` и `ClassD`, который наследуется от класса `ClassA`. В результате выполнения следующего кода:

```
ClassA obj1 = new ClassA();
ClassD obj2 = obj1 as ClassD;
```

`obj2` будет присвоен `null`.

Однако если переменная типа `ClassA` содержит экземпляр переменной типа `ClassD`, то этот оператор будет работать:

```
ClassD obj1 = new ClassD();
ClassA obj2 = obj1;
ClassD obj3 = obj2 as ClassD;
```

В данном случае в переменной `obj3` вместо `null` будет содержаться ссылка на тот же самый объект, что и в переменной `obj1`.

Это делает оператор `as` очень полезным, поскольку выполнение следующего кода приведет к возникновению исключительной ситуации:

```
ClassA obj1 = new ClassA();
ClassD obj2 = (ClassD)obj1;
```

А вот выполнение аналогичного кода с использованием оператора `as` приведет к тому, что переменной `obj2` будет просто присвоено значение `null`. Вследствие этого код, аналогичный приведенному ниже (в нем используются два класса, созданных нами ранее: класс `Animal` и класс `Cow`, производный от класса `Animal`), является совершенно обычным для приложений на C#:

```
public void MilkCow(Animal myAnimal)
{
   Cow myCow = myAnimal as Cow;
   if (myCow != null)
   {
      myCow.Milk();
   }
   else
   {
      Console.WriteLine("{0} не является коровой и не может доиться.",
            myAnimal.Name);
   }
}
```

Это оказывается намного проще, чем осуществлять проверку на предмет возникновения исключительных ситуаций!

Глубокое копирование

В главе 9 рассказывалось о неглубоком копировании с помощью защищенного метода System.Object.MemberwiseClone(), например, следующим способом:

```
public class Cloner
{
   public int Val;

   public Cloner(int newVal)
   {
      Val = newVal;
   }

   public object GetCopy()
   {
      return MemberwiseClone();
   }
}
```

Предположим, однако, что у нас имеются поля, которые представляют собой ссылочные, а не значимые 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 данном случае неглубокая копия будет создана, однако у GetCopy() будет иметься поле, которое ссылается на тот же самый объект, на который ссылается копируемый объект.

Это может быть продемонстрировано с помощью следующего кода:

```
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);
```

Четвертая строка кода, в которой значение присваивается mySource.MyContent.Val — общему полю Val общего поля MyContent исходного объекта, изменяет также и значение myTarget.MyContent.Val. Это происходит потому, что mySource.MyContent ссылается на тот же самый экземпляр объекта, что и myTarget.MyContent. Поэтому исполнение приведенного выше кода приведет к следующим результатам:

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 2
```

Для того чтобы избежать возникновения подобной ситуации, следует воспользоваться глубоким копированием. Мы можем просто внести соответствующие изменения в метод `GetCopy()`, который использовался в предыдущем примере, однако более предпочтительным является стандартный подход, принятый в .NET Framework. Для этого нам потребуется реализовать интерфейс `ICloneable`, у которого имеется единственный метод `Clone()`. У названного метода нет параметров, а возвращает он значение типа объекта, что делает его сигнатуру идентичной использовавшемуся выше методу `GetCopy()`.

Несколько изменив созданные нами классы, мы можем написать следующий код, выполняющий глубокое копирование:

```
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 данном случае мы создали новый объект `Cloner`, используя для этого поле `Val` объекта `Content`, содержащегося в исходном объекте `Cloner` (`MyContent`). Это поле имеет ссылочный тип, поэтому более глубокого копирования не требуется.

Воспользовавшись программой, аналогичной той, которая использовалась для тестирования поверхностного копирования, и заменив в ней метод `GetCopy()` методом `Clone()`, мы получим следующие результаты:

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 5
```

На этот раз внутренние объекты являются независимыми друг от друга.

Обратите внимание, что в некоторых случаях — для более сложных систем объектов — могут потребоваться рекурсивные вызовы метода `Clone()`. Например, если для поля `MyContent` класса `Cloner` также необходимо выполнить глубокое копирование, то нам может понадобиться следующий код:

```
public class Cloner : ICloneable
{
   public Content MyContent = new Content();

   . . .

   public object Clone()
   {
      Cloner clonedCloner = new Cloner();
      clonedCloner.MyContent = MyContent.Clone();
      return clonedCloner;
   }
}
```

В данном случае нами используется конструктор по умолчанию — для упрощения синтаксиса, необходимого для создания нового объекта `Cloner`. Для того чтобы этот код заработал, нам также потребуется реализовать интерфейс `ICloneable` в классе `Content`.

Усовершенствование CardLib, часть 3

Давайте применим полученные знания на практике, реализовав возможность копирования `Card`, `Cards` и `Deck` с использованием интерфейса `ICloneable`. Это может оказаться полезным в некоторых карточных играх, где требуются две колоды карт, которые совершенно не обязательно должны ссылаться на одни и те же объекты `Card` (хотя можно предположить, что нам понадобится другая колода, в которой порядок следования карт должен быть точно таким же, как и в первой).

Реализация функциональной возможности создания копий для класса `Card` в `ChllCardLib` оказывается простой, поскольку вполне достаточно поверхностного копирования (в классе `Card` содержатся только данные значимого типа в виде полей). Нам потребуется только внести следующие изменения в определение класса:

```
public class Card : ICloneable
{
   public object Clone()
   {
      return MemberwiseClone();
   }
```

Обратите внимание, что данная реализация интерфейса `ICloneable` предоставляет просто поверхностное копирование. Никаких правил, определяющих, что именно должно происходить в методе `Clone()`, не существует, и это вполне подходит для наших целей.

Теперь нам необходимо реализовать интерфейс `ICloneable` для класса семейства `Cards`. Это оказывается несколько сложнее, поскольку данная процедура включает в себя создание копий всех объектов `Card` исходного семейства и, следовательно, возникает необходимость глубокого копирования:

```
public class Cards : CollectionBase, ICloneable
{
   public object Clone()
   {
      Cards newCards = new Cards();
      foreach (Card sourceCard in List)
      {
         newCards.Add((Card)sourceCard.Clone());
      }
      return newCards;
   }
```

Наконец, нам понадобится реалнзовать интерфейс `ICloneable` для класса `Deck`. Здесь возникает небольшая проблема: в классе `Deck` отсутствует какой-либо способ изменения содержащихся в нем карт, кроме тасования колоды. Например, не существует способа, позволяющего внести изменения в экземпляр класса `Deck`, расположив в нем карты заданным образом. Для того чтобы обойти эту проблему, мы определяем для класса `Deck` новый частный конструктор, разрешающий передавать определенное семейство `Cards` при создании экземпляра объекта `Deck`. В этом случае код, который позволит создавать копии данного класса, будет иметь следующий вид:

```
public class Deck : ICloneable
{
   public object Clone()
   {
      Deck newDeck = new Deck((Cards)cards.Clone());
      return newDeck;
   }

   private Deck(Cards newCards)
   {
      cards = newCards;
   }
}
```

Для тестирования всех этих изменений мы вновь можем воспользоваться простой клиентской программой (как и прежде, для осуществления тестирования она должна быть помещена в метод `Main()` клиентского проекта):

```
Deck deck1 = new Deck();
Deck deck2 = (Deck)deck1.Clone();
Console.WriteLine("The first card in the original deck is: {0}",
                                  //Первая карта в исходной колоде://
      deck1.GetCard(0));
Console.WriteLine("The first card in the cloned deck is: {0}",
                                  //Первая карта в полученной копии колоды://
      deck2.GetCard(0));
deck1.Shuffle();
Console.WriteLine("Original deck shuffled.");
                                  //Исходная колода перетасована.//
Console.WriteLine("The first card in the original deck is: {0}",
                                  //Первая карта в исходной колоде://
      deck1.GetCard(0));
Console.WriteLine("The first card in the cloned deck is: {0}",
                                  //Первая карта в полученной копии колоды://
      deck2.GetCard(0));
```

В этом случае выходной поток будет иметь примерно следующий вид:



Здесь необходимо сделать одно важное замечание: все карточные правила, определяемые статическими членами класса `Card`, применимы ко всем объектам `Card` для любой колоды `Deck`. Невозможно иметь две колоды карт, подчиняющиеся разным правилам. Для данной библиотеки классов такое положение дел вполне приемлемо; если же некоторому приложению потребуются собственные правила, оно сможет поддерживать их самостоятельно, например, за счет установки значений статических членов класса `Card` каждый раз, когда происходит переход от использования одной колоды к использованию другой.

Исключительные ситуации, определяемые программистом

Выше обсуждались исключительные ситуации и то, каким образом можно использовать блоки `try...catch...finally` для их обработки. Вы также познакомилнсь с несколькими стандартными для .NET Framework исключительными ситуациями, в том числе с `System.Exception` — базовым классом для исключительных ситуаций. В некоторых случаях вместо того, чтобы применять стандартные исключительные ситуации, полезно создавать и использовать в приложениях свои собственные классы исключительных ситуаций, производные от этого базового класса. Это позволяет более точно определять информацию, которая отправляется коду, осуществляющему перехват этой ситуации, а также использовать более специфические программы для обработки конкретных исключительных ситуаций. Существует, например, возможность добавить в такой класс исключительных ситуаций новое свойство, которое сделает возможным доступ к какой-либо информации более низкого уровня, что позволит программе-обработчику исключительных ситуаций внести соответствующие изменения или просто получить дополнительную информацию о причинах возникновения данной исключительной ситуации.



После того как класс исключительных ситуаций определен, появляется возможность внести его в список исключительных ситуаций, распознаваемых VS с помощью пункта меню **Debug | Exceptions...**. Представленное на рисунке слева диалоговое окно позволяет управлять реакцией VS на возникновение необрабатываемой исключительной ситуации данного типа. Здесь можно настроить VS либо на прекращение выполнения программы и запуск отладчика, либо на продолжение выполнения программы: во-первых, когда возникает исключительная ситуация такого рода и, во-вторых, если эта ситуация не обрабатывается.

Усовершенствование CardLib, часть 4

Применение исключительных ситуаций, определяемых пользователем, лучше всего может быть проиллюстрировано внесением изменений в проект `ChlCardLib`. В настоящий момент метод `Deck.GetCard()` использует стандартную исключительную ситуацию .NET, однако мы внесем соответствующие изменения с тем, чтобы он мог использовать исключительную ситуацию, определяемую программистом.

Во-первых, необходимо определить исключительную ситуацию. Это достигается посредством определения нового класса в новом файле `Exceptions.cs`:

```
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.")
    {
       deckContents = sourceDeckContents;
    }
}
```

Названный класс должен создаваться при создании экземпляра класса `Cards`. Он позволяет осуществлять доступ к данному объекту `Cards` посредством свойства `DeckContents` и обеспечивает передачу подходящего сообщения об ошибке базовому конструктору `Exception`, после чего оно становится доступным через свойство `Message` данного класса.

Далее мы добавляем код, передающий эту исключительную ситуацию в `Deck.cs` (вместо использовавшейся ранее стандартной исключительной ситуации):

```
public Card GetCard(int cardNum)
{
   if (cardNum >= 0 && cardNum <= 51)
      return cards[cardNum];
   else
      throw new CardOutOfRangeException((Cards)cards.Clone());
}
```

В качестве начального значения свойству `DeckContents` присваивается глубокая копия текущего содержимого объекта `Deck`, которое представляет собой объект `Cards`. Это означает, что мы можем узнать, какой вид имело содержимое в момент возникновения исключительной ситуации, благодаря чему последующие изменения содержимого колоды не приведут к потере информации.

Для тестирования этих изменений можно воспользоваться следующим клиентским кодом:

```
Deck deck1 = new Deck();
try
{
   Card myCard = deck1.GetCard(60);
}
catch (CardOutOfRangeException e)
{
   Console.WriteLine(e.Message);
   Console.WriteLine(e.DeckContents[0]);
}
```

В результате его выполнения будет получено следующее:



В данном случае код, ответственный за перехват исключительной ситуации, вывел свойство исключительной ситуации `Message` на экран. Мы также выводим первую карту, хранящуюся в объекте `Cards`, которую мы получаем через `DeckContents`, — просто для того, чтобы доказать, что мы имеем доступ к семейству `Cards` через определенный нами объект исключительной ситуации.

Итоги

В настоящей главе было рассмотрено множество способов, которые могут использоваться для того, чтобы существенно увеличить мощь создаваемых с применением ООП приложений и сделать их более интересными. И хотя использование этих способов не требует особых усилий, они позволяют значительно облегчить работу с различными классами и, соответственно, упростить написание последующего кода.

Каждая из рассмотренных тем находит множество применений. Вероятность того, что вам почти в каждом приложении придется сталкиваться с семействами в той или иной форме, очень велика, и создание строго типизированных семейств при необходимости организовать работу с группой однотипных объектов может значительно упростить вам жизнь. Теперь вы знаете, как в таких семействах можно использовать индексаторы, которые облегчают доступ к объектам, входящим в эти семейства.

Вы также изучили перегрузку операторов, которая позволяет определять, каким образом операторы, например, + и –, могут работать с нашими классами; кроме того, вы познакомились с глубоким копированием, которое играет очень важную роль, позволяя избегать одной из наиболее распространенных в ООП ошибок — непреднамеренной передачи пользователю ссылок на внутренние данные. Наконец, вы узнали, как можно описывать свои собственные объекты исключительных ситуаций и передавать более подробную информацию программе-обработчику исключительной ситуации.

В следующей главе будет рассмотрен последний — и чрезвычайно важный — аспект ООП в .NET: события.

Упражнения

1. Создайте класс семейства `People` (люди), который представляет собой семейство класса `Person` (лицо), приведенного ниже. Доступ к элементам семейства должен быть организован через строковый индексатор, являющийся именем лица, которое идентично свойству `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 класс `Person` из предыдущего упражнения, перегрузив операторы >, <, >= и <= так, чтобы они выполняли сравнение свойств `Age` экземпляров класса `Person`.

3. Добавьте в класс `People` метод `GetOldest()`, который будет возвращать массив объектов `Person` с наибольшим значением свойства `Age` (этот массив состоит из одного и более объектов, поскольку значение данного свойства может оказаться одинаковым у нескольких элементов). Для этого воспользуйтесь перегруженными операторами, определенными выше.

4. Реализуйте в классе `People` интерфейс `ICloneable`, который бы обеспечивал выполнение глубокого копирования.



12

События

В этой последней главе, посвященной ООП, будет обсуждаться один из наиболее часто встречающихся в .NET приемов ООП — использование **событий**.

Обсуждение начнется с основ — с вопроса о том, что на самом деле представляют из себя события. После этого будет рассмотрено применение некоторых простых событий на практике. Изучив этот вопрос, мы познакомимся с тем, каким образом можно создавать и использовать собственные события.

Во второй части главы будет усовершенствована библиотека классов `CardLib` с помощью включения в нее события. Кроме того — поскольку это наша последняя остановка перед знакомством с более сложными темами — мы позволим себе немного развлечься и создадим приложение для карточных игр, которое будет использовать созданную нами библиотеку классов.

Что такое событие?

События похожи на исключительные ситуации в том смысле, что они **создаются** (генерируются) объектами, и у нас имеется возможность задать код, который будет выполняться при их наступлении. Между ними, однако, существует несколько важных отличий. Наиболее существенным является отсутствие конструкции, предназначенной для обработки событий и эквивалентной конструкции `try...catch`. Вместо этого на события необходимо подписываться. Подписаться на некоторое событие — означает указать код, который должен выполняться при наступлении данного события и который имеет форму **обработчика событий**.

У события может быть несколько обработчиков, приписанных к нему, и в случае наступления события все они будут вызваны. В их числе могут быть обработчики событий, которые является частью класса, к которому принадлежит объект, сгенерировавший событие, однако с тем же успехом обработчики событий могут находиться и в других классах.

Сами по себе обработчики событий являются простыми функциями. Единственное ограничение, налагаемое на функцию — обработчика событий заключается в том, что ее сигнатура (возвращаемое значение и параметры) должна соответствовать обрабатываемому событию. Эта сигнатура является составной частью определения события и задается с помощью **делегата**.

> *Именно благодаря тому, что делегаты используются при обработке событий, они оказываются такими полезными; именно по этой причине мы посвятили некоторое время их изучению в главе 6; возможно, вы пожелаете перечитать*

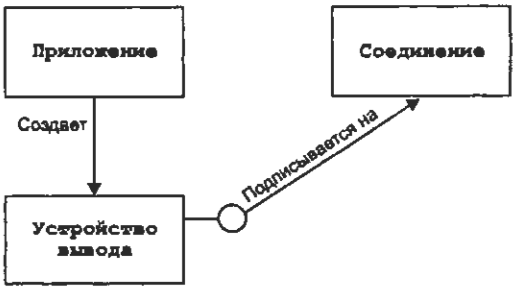
тот раздел, чтобы освежить в памяти информацию о том что представляют собой делегаты и каким образом они используются.

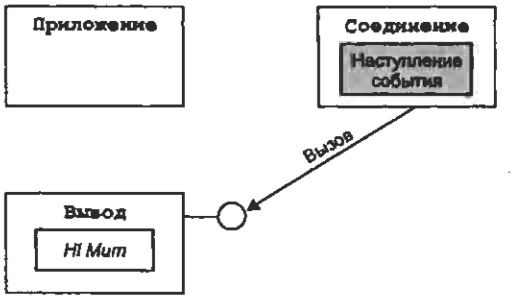
Последовательность действий при обработке событий примерно такова:

Во-первых, в приложении создается объект, который может генерировать некоторое событие. В качестве примера рассмотрим приложение, которое занимается немедленной отправкой сообщений, а создаваемый им объект представляет собой соединение с удаленным пользователем (см. рис. слева). Этот объект мог бы сгенерировать событие, когда, например, по данному соединению приходит сообщение от удаленного пользователя.



Затем приложение подписывается на событие. Приложение для немедленной отправки сообщений может добиться этого путем определения функции, допускающей использование с типом делегата, задаваемым данным событием, и передающей событию ссылку на эту функцию. В качестве такой функции-обработчика события может использоваться метод или какой-либо другой объект — скажем, объект, представляющий некоторое устройство вывода сообщений, предназначенное для немедленного вывода сообщений при их поступлении (см. рис. справа).





Когда возникает некоторое событие, подписавшийся на него обработчик получает уведомление. При поступлении сообщения через соединение осуществляется вызов метода — обработчика события на объект устройства вывода. Поскольку здесь применяется стандартный метод, то объект, сгенерировавший событий, имеет возможность передать любую относящуюся к данному событию информацию посредством параметров, что делает аппарат событий весьма гибким. В данном примере одним из параметров может быть текст, используемый для вывода сообщения посредством обработчика события на объект, представляющий устройство вывода (см. рис. слева).

Использование событий

В этом разделе сначала будет рассматриваться код, требующийся для обработки событий, а затем вопрос о том, каким образом программист может определять и использовать свои собственные события.

Обработка событий

Чтобы обработать событие, на него необходимо подписаться путем задания функции-обработчика этого события, причем сигнатура этой функции должна совпадать с сигнатурой, которая задается делегатом, предназначенным для работы с данным событием. В качестве примера рассмотрим простой объект — таймер, генерирующий события, в результате наступления которых вызывается функция-обработчик.

Практикум: обработка событий

1. Создайте новое консольное приложение с именем `Ch12Ex01` в директории `C:\BegCSharp\Chapter12\`.

2. Модифицируйте код в `Class1.cs` следующим образом:

```
using System;
using System.Timers;

namespace Ch12Ex01
{
   class Class1
   {
      static int counter = 0;

      static string displayString =
                        "This string will appear one letter at a time. ";

      static void Main(string[] args)
      {
         Timer myTimer = new Timer(100);
         myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
         myTimer.Start();
         Console.ReadLine();
      }

      static void WriteChar(object source, ElapsedEventArgs e)
      {
         Console.Write(displayString[counter++ % displayString.Length]);
      }
   }
}
```

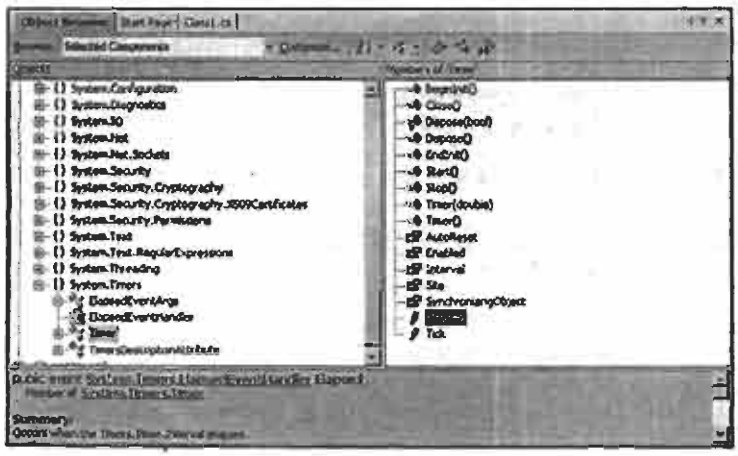
3. Запустите приложение (нажатие клавиши *Enter* в процессе работы приложения приведет к прекращению его выполнения):



Как это работает

Объект, используемый для событий, представляет собой экземпляр класса `System.Timers.Timer`. При инициализации этого объекта задается интервал времени (в миллисекундах). После того как объект `Timer` запускается посредством вызова его метода `Start()`, генерируется последовательность событий, отстоящих друг от друга во времени на заданную величину интервала. Функция `Main()` инициализирует объект `Timer` с временным интервалом, равным 100 миллисекундам, поэтому после запуска он генерирует события с частотой 10 раз в секунду:

```
static void Main(string[] args)
{
   Timer myTimer = new Timer(100);
```



Объект Timer обладает событием, которое называется Elapsed (истек интервал), что можно увидеть, воспользовавшись окном Object Browser, представленным на рисунке слева.

Сигнатура, необходимая для обработчика рассматриваемого вида событий, соответствует типу делегата System.Timers.ElapsedEventHandler, который является одним из стандартных делегатов, определенных в .NET Framework. Этот делегат может использоваться для функций, обладающих следующей сигнатурой:

```
void functionName(object source, ElapsedEventArgs e);
```

Объект Timer передает ссылку на самого себя в первом параметре и экземпляр объекта ElapsedEventArgs — во втором. На данном этапе можно не обращать на эти параметры никакого внимания — позже мы вернемся к их рассмотрению.

В нашем коде имеется метод с совпадающей сигнатурой:

```
static void WriteChar(object source, ElapsedEventArgs e)
{
   Console.Write(displayString[counter++ % displayString.Length]);
}
```

Данный метод использует два статических поля класса Class1 — counter и displayString — для вывода отдельного символа. При каждом вызове этого метода будет выводиться новый символ.

Следующая задача — привязать этот обработчик к событию, т. е. подписаться на событие. С этой целью используется оператор +=, который позволяет включить в событие обработчик в виде нового экземпляра делегата, инициализированного с используемым методом обработчика событий:

```
static void Main(string[] args)
{
   Timer myTimer = new Timer(100);
   myTimer.Tick += new EventHandler(WriteChar);
```

Эта команда (где используется присущий делегатам странный на вид синтаксис) добавляет обработчик в список обработчиков, вызываемых при возникновении события Elapsed. Можно добавлять в этот список сколь угодно много обработчиков, при этом все они должны удовлетворять требуемому критерию. Каждый из этих обработчиков будет вызываться при наступлении события поочередно.

Все, что осталось необходимым выполнить в функции Main(), — это запустить таймер:

```
   myTimer.Start();
```

Поскольку не нужно, чтобы приложение окончилось до того, как будет обработано хотя бы одно событие, то следует приостановить выполнение функции Main(). Простейшим способом такой приостановки является ожидание ввода пользователя, так как выполнение этой команды не сможет завершиться ранее, чем пользователь введет строку текста и/или нажмет клавишу *Enter*.

```
   Console.ReadLine();
}
```

Хотя выполнение метода `Main()` фактически оказывается остановленным, объект `Timer` продолжает функциоинровать. При наступлении очередного события он вызывает описанный метод `WriteChar()`, который выполняется параллельно с выполненнем оператора `Console.ReadLine()`.

Определение событий

В качестве примера определения и использования событий нужно реализовать версию программы немедленной отправки сообщений, рассмотренную в начале этой главы, и создать объект `Connection` (соединенне), который будет генерировать события, обрабатывающиеся объектом `Display` (вывод).

Практикум: Определение событий

1. Создайте новое консольное приложение с именем `Ch12Ex02` в директорин `C:\BegCSharp\Chapter12\`.

2. Добавьте новый класс — `Connection` — в файл `Connection.cs`:

```
using System;
using System.Timers;

namespace Ch12Ex02
{
   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 void CheckForMessage(object source, ElapsedEventArgs e)
      {
         Console.WriteLine("Checking for new messages.");
         Random random = new Random();
         if ((random.Next(9) == 0) && (MessageArrived != null))
         {
            MessageArrived("Hello Mum!");
         }
      }
   }
}
```

3. Добавьте новый класс — Display — в файл Display.cs:

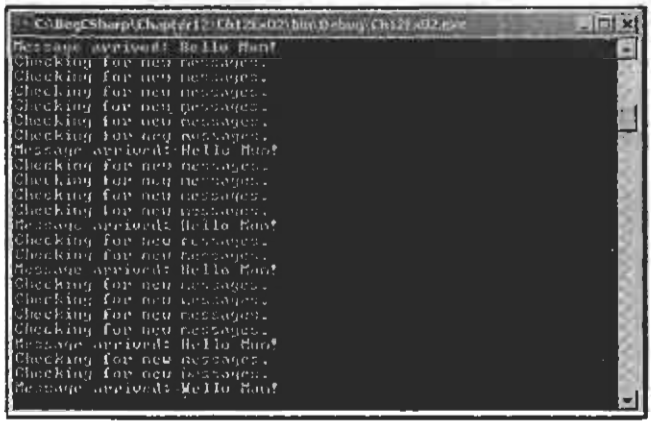
```
using System;

namespace Ch12Ex02
{
   public class Display
   {
      public void DisplayMessage(string message)
      {
         Console.WriteLine("Message arrived: {0}", message);
      }
   }
}
```

4. Внесите следующие изменения в файл Class1.cs:

```
using System;

namespace Ch12Ex02
{
   class Class1
   {
      static void Main(string[] args)
      {
         Connection myConnection = new Connection();
         Display myDisplay = new Display();
         myConnection.MessageArrived +=
                  new MessageHandler (myDisplay.DisplayMessage);
         myConnection.Connect();
         Console.ReadLine();
      }
   }
}
```



5. Запустите приложение (см. рис. слева).

Как это работает

Класс, который выполняет львиную долю работы в данном приложении,— это класс Connection. Объект Timer используется экземплярами этого класса во многом аналогично тому, как он использовался в первом приведенном в данной главе примере, т. е. инициализируется конструктором класса и предоставляет доступ к информации о его состоянии (включен или выключен) через функции 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();
    }

    . . .
}
```

Кроме того, в конструкторе регистрируется обработчик событий для события `Elapsed` — точно так же, как это делалось в первом примере. Метод обработчика — `CheckForMessage()` — будет генерировать событие в среднем один раз на десять вызовов. Но прежде чем перейти к рассмотрению соответствующего кода, следует познакомиться с собственно определением события.

Прежде чем определять событие, необходимо определить тип делегата, который будет использоваться с данным событием, т. е. позволяющий определить, какой именно сигнатуре должен соответствовать метод, выполняющий обработку события. Это достигается за счет использования стандартного синтаксиса описания делегата, который определяется как общий в пространстве имен `Ch11Ex02`, для того чтобы обеспечить доступ к нему из внешнего кода:

```
namespace Ch12Ex02
{
   public delegate void MessageHandler(string messageText);
```

Этот тип делегата, которому присвоено имя `MessageHandler`, определяет сигнатуру функции типа `void`, обладающей одним параметром типа `string`. Этот параметр может быть использован для передачи объекту `Display` сообщения, полученного от объекта `Connection`.

После описания делегата (либо использования подходящего делегата, описанного ранее) появляется возможность определить собственно событие как член класса `Connection`:

```
public class Connection
{
   public event MessageHandler MessageArrived;
```

Далее событию присваивается имя (в данном случае — имя `MessageArrived`), после чего это событие объявляется с помощью ключевого слова `event` и указывается тип делегата, который необходимо использовать (`MessageHandler`, определенный ранее).

После того как событие определено подобным способом, появляется возможность вызывать его наступление простым обращением по имени — точно таким же образом, как если бы оно представляло собой метод с сигнатурой, определяемой делегатом. Например, вызвать наступление данного события можно следующим образом:

```
MessageArrived("This is a message.");
```

Если бы делегат был описан как не имеющий параметров, то можно было бы вызвать наступление события проще:

```
MessageArrived();
```

Напротив, можно было бы описать большее число параметров, что потребовало бы большего количества кода для генерации события.

В данном случае метод `CheckForMessage()` (проверка, не пришло ли сообщение) имеет следующий вид:

```
private void CheckForMessage(object source, ElapsedEventArgs e)
{
   Console.WriteLine("Checking for new messages.");
   Random random = new Random();
   if ((random.Next(9) == 0) && (MessageArrived != null))
   {
      MessageArrived("Hello Mum!");
   }
}
```

Здесь используется экземпляр объекта класса `Random`, который описывался в предшествующих главах. Он генерирует случайное число в диапазоне между 0 и 9, и наступление события вызывается только в том случае, если это число равно 0, что должно происходить в 10% случаев. Это позволяет эмулировать опрос соединения на предмет поступления сообщения, которое не обязательно приходит каждый раз при выполнении проверки.

Обратите внимание на то, что здесь используется некоторая дополнительная логика. Наступление события вызывается только в том случае, если значение выражения `MessageArrived != null` равняется `true`. Это выражение, в котором также используется синтаксис делегата, хотя и в несколько необычном виде, означает: "Имеются ли у данного события какие-либо подписчики?". Если подписчиков не существует, то `MessageArrived` получает значение `null`, и в этом случае нет никакого смысла генерировать наступление события.

Класс, являющийся подписчиком данного события, называется `Display` и состоит из единственного метода `DisplayMessage()`, который описывается следующим образом:

```
public class Display
{
   public void DisplayMessage(string message)
   {
      Console.WriteLine("Message arrived: {0}", message);
   }
}
```

Сигнатура этого метода соответствует сигнатуре, описываемой делегатом данного типа (и являющейся общей, что представляет собой необходимое условие для обработчиков событий во всех классах, отличных от класса, в котором генерируется данное событие), поэтому данный метод может использоваться для реакции на наступление события `MessageArrived`.

Все, что теперь необходимо сделать в методе `Main()`,— это инициализировать экземпляры классов `Connection` и `Display` и запустить процесс. Код, который для этого нужно использовать, аналогичен коду 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.ReadLine();
}
```

В данном случае, после того как аппарат работы с событиями запущен посредством обращения к методу `Connect()` объекта `Connection`, для приостановки выполнения функции `Main()` вызывается метод `Console.ReadLine()`.

Многоцелевые обработчики событий

Сигнатура, используемая для события `Timer.Elapsed`, содержит два параметра, тип которых очень часто встречается в различных обработчиках событий. Вот эти параметры:

- `object source` — ссылка на объект, который вызвал наступление события
- `ElapsedEventArgs e` — параметры, передаваемые событием

Причина, по которой в данном событии — да и во многих других событиях тоже — используется параметр типа `object`, заключается в том, что очень часто необходимо использовать один и тот же обработчик событий для обработки нескольких одинаковых событий, генерируемых различными объектами, и при этом отличать, каким именно объектом сгенерировано данное событие.

Чтобы пояснить и проиллюстрировать это, следует несколько расширить предыдущий пример.

Практикум: Использование многоцелевого обработчика событий

1. Создайте новое консольное приложение с именем `Ch12Ex03` в директории `C:\BegCSharp\Chapter12\`.
2. Скопируйте код в файлы `Class1.cs`, `Connection.cs` и `Display.cs` из соответствующих файлов в `Ch12Ex02` и убедитесь в том, что во всех файлах изменены пространства имен с `Ch12Ex02` на `Ch12Ex03`.
3. Добавьте новый класс — `MessageArrivedEventArgs`, который находится в файле `MessageArrivedEventArgs.cs`:

```
using System;

namespace Ch12Ex03
{
   public class MessageArrivedEventArgs : EventArgs
   {
      private string message;

      public string Message
      {
         get
         {
            return message;
         }
      }

      public MessageArrivedEventArgs()
      {
         message = "No message sent.";
      }

      public MessageArrivedEventArgs(string newMessage)
      {
         message = newMessage;
      }
   }
}
```

4. Внесите следующие изменения в файл Connection.cs:

```
namespace Ch12Ex03
{
   public delegate void MessageHandler(Connection source,
                                       MessageArrivedEventArgs e);

   public class Connection
   {
      public event MessageHandler MessageArrived;

      private string name;

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

      . . .

      private void CheckForMessage(object source, EventArgs e)
      {
         Console.WriteLine("Checking for new messages.");
         Random random = new Random();
         if ((random.Next(9) == 0) && (MessageArrived != null))
         {
            MessageArrived(this, new MessageArrivedEventArgs("Hello Mum!"));
         }
      }

      . . .

   }
}
```

5. Внесите следующие изменения в файл Display.cs:

```
public void DisplayMessage(Connection source, MessageArrivedEventArgs e)
{
   Console.WriteLine("Message arrived from: {0}", source.Name);
   Console.WriteLine("Message Text: {0}", e.Message);
}
```

6. Внесите следующие изменения в файл Class1.cs:

```
static void Main(string[] args)
{
   Connection myConnection1 = new Connection();
   myConnection1.Name = "First connection.";
   Connection myConnection2 = new Connection();
   myConnection2.Name = "Second connection.";
   Display myDisplay = new Display();
   myConnection1.MessageArrived +=
            new MessageHandler(myDisplay.DisplayMessage);
   myConnection2.MessageArrived +=
            new MessageHandler(myDisplay.DisplayMessage);
   myConnection1.Connect();
   myConnection2.Connect();
   Console.ReadLine();
}
```



7. Запустите приложение (см. рис. слева).

Как это работает

Передавая ссылку на объект, который вызвал наступление события в качестве одного из параметров обработчика событий, мы получаем возможность настраивать реакцию обработчика событий на конкретные объекты. Эта ссылка обеспечивает возможность доступа к исходному объекту, включая его свойства.

Передавая параметры, содержащиеся в классе, который наследуется от `System.EventArgs` (как, например, `ElapsedEventArgs`), мы получаем возможность передавать в качестве параметров (как, например, параметр `Message` в классе `MessageArrivedEventArgs`) любую дополнительную информацию.

Кроме того, такие параметры позволяют использовать преимущество полиморфизма. Можно было бы определить обработчик события `MessageArrived` следующим способом:

```
public void DisplayMessage(object source, EventArgs e)
{
   Console.WriteLine("Message arrived from: {0}",
                     ((Connection)source).Name);
   Console.WriteLine("Message Text: {0}",
                     ((MessageArrivedEventArgs)e).Message);
}
```

и видоизменить определение делегата в файле `Connection.cs` таким образом:

```
public delegate void MessageHandler(object source, EventArgs e);
```

Приложение будет выполняться в точности так же, как оно выполнялось и ранее, однако в этом случае функция `DisplayMessage()` стала более гибкой (по крайней мере в теории — для получения естественного продукта потребуется дополнительная реализация). Тот же самый обработчик событий мог бы работать и с другими событиями, такими как `Timer.Elapsed`, хотя для этого потребуется внести некоторые дополнительные изменения в тело обработчика, которые позволят ему должным образом обрабатывать параметры, передаваемые при наступлении события (в результате приведения их типа к типу объектов `Connection` и `MessageArrivedEventArgs` в данном случае возникнет исключительная ситуация и вместо этого следует использовать оператор `as`).

Прежде чем продолжить движение вперед, следует обратить внимание на то, что результатом выполнения данного примера будут, скорее всего, пары событий, сгенерированных двумя объектами `Connection`. Такая ситуация возникает благодаря способу, с помощью которого происходит генерирация случайных чисел классом `Random`. В момент создания экземпляра объекта типа `Random` используется некоторое **начальное значение**. Оно применяется для получения последовательности псевдослучайных чисел с помощью некоторой сложной функции. (Компьютеры не имеют возможности создавать действительно случайные числа.) Можно задавать начальное значение в конструкторе экземпляра `Random`, однако если он не используется (как в настоящем примере), то в качестве начального значения используется текущее время. Поскольку мы создаем экземпляры и начинаем использовать оба наших объекта `Connection` в последовательных строках кода, то это с большой

вероятностью может привести к тому, что при реакции на наступление событий Timer.Elapsed ими будут использоваться один и те же начальные значения, поэтому в том случае, когда один из объектов отправляет свое сообщение, второй, вероятнее всего, будет отправлять свое. Это прямое следствие высокой скорости обработки инструкций, и, для того чтобы обойти эту ситуацию, необходимо задавать начальные значения каким-то другим способом. Одно из возможных решений, которое здесь не будет реализовываться, заключается в том, чтобы использовать единственный экземпляр класса Random, доступный из обоих объектов Connection. В таком случае оба этих объекта будут использовать одну и ту же последовательность случайных чисел, поэтому одновременная отправка сообщений окажется маловероятной.

Возвращаемые значения и обработчики событий

Все обработчики событий, рассматриваемые до настоящего момента, обладали возвращаемым типом void. Существует возможность задавать для события тип возвращаемого значения, однако это может привести к некоторым проблемам, которые связаны с тем, что наступление некоторого события может приводить к вызову нескольких обработчиков. Если каждый из вызванных обработчиков будет возвращать какое-либо значение, то становится непонятным, какое именно возвращаемое значение будет получено.

В системе эта проблема решается таким образом, что доступным оказывается только последнее возвращаемое обработчиком значение, и это будет значение, возвращаемое обработчиком, который последним подписался на данное событие.

Вероятно, существуют ситуации, когда такая функциональная возможность окажется полезной, хотя авторам так и не удалось придумать ни одной такой ситуации. Рекомендуется использовать обработчики событий типа void и избегать параметры типа out.

Расширение и использование CardLib

Теперь следует внести некоторые дополнения в библиотеку классов, созданную в предшествующей главе,— Ch11CardLib. Будет использоваться проект по созданию библиотеки классов с именем Ch12CardLib, в котором изначально находится код, идентичный коду в Ch11CardLib (не считая того, что в нем используются имена из пространства имен Ch12CardLib вместо Ch11CardLib).

Событие, которое мы собираемся включить в библиотеку, будет генерироваться в тот момент, когда последний объект Card получается в объекте Deck с помощью метода GetCard(), после чего инициируется наступление события LastCardDrawn (последняя сданная карта). Это событие позволит подписчикам автоматически перетасовывать колоду, уменьшая объем вычислений, которые приходится выполнять клиенту. Делегат, определенный для данного события (LastCardDrawnHandler), должен передавать ссылку на соответствующий объект Deck, для того чтобы обработчик мог иметь доступ к методу Shuffle() независимо от своего местонахождения:

```
namespace Ch12CardLib
{
    public delegate void LastCardDrawnHandler(Deck currentDeck);
```

Код, который используется для определения и генерирования события, весьма 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());
}
```

Вот и весь код, который потребовался для включения события в определение класса `Deck`. Теперь осталось только воспользоваться им.

Клиентская программа карточной игры для CardLib

Позор, если, потратив столько времени на разработку библиотеки `CardLib`, мы так и не воспользуемся ей. Перед тем как завершить раздел, посвященный ООП в C# и .NET Framework, самое время немного поразвлечься и написать основу для приложения, которое использует хорошо знакомые нам классы, связанные с игральными картами.

Как и в предшествующих главах, мы добавим клиентское консольное приложение в `Ch12CardLib`, добавим ссылку на проект `Ch12CardLib` и объявим этот проект начальным. Это приложение мы назовем `Ch12CardClient`.

Для начала создадим новый класс, который мы назовем `Player` (игрок), в новом файле в приложении `Ch12CardClient` — `Player.cs`. В нем будет содержаться частное поле `Cards` с именем `hand` (рука), частное строковое поле с именем `name` и два поля, доступных в режиме "только чтение": `Name` и `PlayHand`. Эти поля используются для представления частных полей. Обратите внимание на то, что, хотя свойство `PlayHand` доступно в режиме "только чтение", у нас будет доступ на запись по ссылке на возвращаемое поле `hand`, что позволит изменять карты, находящиеся на руках у данного игрока.

Мы также спрячем конструктор по умолчанию, объявив его как частный, и добавим общий конструктор не по умолчанию, которому будут передаваться первоначальные значения свойства `Name` экземпляров класса `Player`.

Наконец, мы предусмотрим метод типа `bool` с именем `HasWon()` (выиграл). Этот метод будет возвращать значение `true`, если все карты, находящиеся на руках у данного игрока, имеют одну масть (довольно простое условие выигрыша, но в данном случае это не играет особой роли).

Код в `Player.cs` имеет следующий вид:

```
using System;
using Ch12CardLib;

namespace Ch12CardClient
{
   public class Player
   {
      private Cards hand;
      private string name;
```

```
        public string Name
        {
            get
            {
                return name;
            }
        }

        public Cards PlayHand
        {
            get
            {
                return hand;
            }
        }

        private Player()
        {
        }

        public Player(string newName)
        {
            name = newName;
            hand = new Cards();
        }

        public bool HasWon()
        {
            bool won = true;
            Suit match = hand[0].suit;
            for (int i = 1; i < hand.Count; i++)
            {
                won &= hand[i].suit == match;
            }
            return won;
        }
    }
}
```

Далее следует определить класс, который будет вести карточную игру. Назовем его `Game` (игра). Он будет размещен в файле `Game.cs` в проекте `Ch12CardClient`.

В качестве членов этого класса предусмотрено наличие четырех частных полей:

- `playDeck` — переменная типа `Deck`, в которой содержится используемая колода карт
- `currentCard` — значение типа `int`, которое используется в качестве указателя на очередную сдаваемую карту
- `players` — массив объектов типа `Player`, которые представляют участников карточной игры
- `discardedCards` — семейство объектов `Cards`, которые уже вышли из игры, но пока еще не возвращены в колоду

Конструктор данного класса по умолчанию инициализирует и перетасовывает колоду `Deck`, находящуюся в `playDeck`, устанавливает указатель `currentCard` на 0 (на первую карту в `playDeck`) и привязывает обработчик событий `Reshuffle()` (пересдача) к событию `playDeck.LastCardDrawn`. Названный обработчик просто перетасовывает колоду, включая карты, находящиеся в `discardedCards`, и подготавливает новую колоду к раздаче, устанавливая указатель `currentCard` на 0.

В классе `Game` содержатся также два полезных метода: `SetPlayers()`, задающий игроков (в виде массива объектов типа `Player`), и `DealHands()`, сдающий карты игрокам (по 7 карт каждому). Число игроков должно находиться в диапазоне от 2 до 7, что гарантирует достаточное количество карт для каждого из них.

Наконец, в файл включается метод `PlayGame()`, в котором содержится логика игры. Мы вернемся к этой функции, после того как рассмотрим код в файле `Class1.cs`. Весь остальной код файла `Game.cs` имеет следующий вид:

```
using System;
using Ch12CardLib;

namespace Ch12CardClient
{
   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();
      }
      private void Reshuffle(Deck currentDeck)
      {
         currentDeck.Shuffle();
         discardedCards = new Cards();
         currentCard = 0;
      }

      public void SetPlayers(Player[] newPlayers)
      {
         if (newPlayers.Length > 7)
            throw new ArgumentException("A maximum of 7 players may play this" +
                                        " game.");
         if (newPlayers.Length < 2)
            throw new ArgumentException("A minimum of 2 players may play this" +
                                        " game.");
         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()
      {
         // Сюда будет включаться дальнейший код.
      }
   }
}
```

В файле `Class1.cs` содержится функция `Main()`, которая инициализирует и запускает игру. Эта функция выполняет следующие шаги:

- На экран выводится заставка.
- Пользователь получает приглашение на ввод числа игроков, которое должно находиться в интервале между двумя и семью.
- В соответствии с введенным числом игроков определяются объекты типа `Player`.
- Каждому пользователю выводится приглашение, запрашивающее его имя, которое будет использоваться при инициализации соответствующего объекта типа `Player` в массиве.
- Создается объект `Game` и с помощью метода `SetPlayers()` определяются игроки, которые будут принимать участие в игре.
- Начинается собственно игра посредством метода `PlayGame()`.
- Возвращаемое значение метода `PlayGame()` типа `int` используется для вывода сообщения о том, кто из игроков выиграл (возвращаемое значение представляет собой индекс победителя в массиве объектов типа `Player`).

Код, выполняющий все эти действия (снабженный комментариями для большей ясности), приводится ниже:

```
static void Main(string[] args)
{
   // Вывод заставки
   Console.WriteLine("KarliCards: a new and exciting card game.");
   Console.WriteLine("To win you must have 7 cards of the same suit in" +
                     " your hand.");
   Console.WriteLine();

   // Приглашение для ввода числа игроков
   bool inputOK = false;
   int choice = -1;
   do
   {
      Console.WriteLine("How many players (2-7)?");
      string input = Console.ReadLine();
      try
      {
         // Попытка преобразовать введенную информацию в корректное
         // число игроков
         choice = Convert.ToInt32(input);
         if ((choice >= 2) && (choice <= 7))
            inputOK = true;
      }
      catch
      {
         // В случае неудачи продолжаем выводить приглашение
      }
   } while (inputOK == false);

   // Инициализация массива объектов Player.
   Player[] players = new Player[choice];
```

```
        // Запрос имен игроков.
        for (int p = 0; p < players.Length; p++)
        {
           Console.WriteLine("Player {0}, enter your name:", p + 1);
           string playerName = Console.ReadLine();
           players[p] = new Player(playerName);
        }

        // Начало игры
        Game newGame = new Game();
        newGame.SetPlayers(players);
        int whoWon = newGame.PlayGame();

        // Вывод информации о выигравшем игроке
        Console.WriteLine("{0} has won the game!", players[whoWon].Name);
    }
```

Теперь обратимся к методу `PlayGame()`, представляющему основное тело приложения. Этот метод не будет описываться подробно, так как он снабжен комментариями, что делает его несколько более понятным. На самом деле, за некоторыми исключениями, в коде нет ничего столь уж сложного.

Игра протекает следующим образом: каждый игрок видит собственные карты и одну открытую карту на столе. Игроки могут взять либо открытую карту, либо новую карту из колоды. Взяв карту, игрок должен сбросить одну карту, положив ее либо на место открытой карты, если им была взята именно открытая карта, либо поверх лежащей на столе (отправляя предыдущую карту, лежащую на столе, в семейство `discardedCards`).

Ключевым моментом, который необходимо иметь в виду при изучении кода, является способ, используемый для манипуляций над объектами типа `Card`. Теперь становится понятным, почему они были описаны как перемеиные ссылочного типа, а не как значимого (посредством использования структуры). Один объект `Card` может находиться одиовременно в нескольких местах, поскольку на него могут ссылаться объект `Deck`, поля `hand` различных объектов `Player`, семейство `discardedCards` и объект `playCard` (поскольку карта в настоящий момент находится на столе). Это значительно упрощает процедуру отслеживания движения карт и используется, в частности, в коде, который отвечает за сдачу очередной карты из колоды. Карта может быть принята только в том случае, еслн она ие находится у какого-нибудь игрока на руках, на столе или в семействе `discardedCards`.

Код имеет следующий вид:

```
    public int PlayGame()
    {
        // Игра происходит только при наличии игроков
        if (players == null)
           return -1;

        // Первоначальная сдача карт.
        DealHands();

        // Инициализация переменных, используемых в карточной игре, включая
        // карту, которая кладется на стол первой: playCard
        bool GameWon = false;
        int currentPlayer;
        Card playCard = playDeck.GetCard(currentCard++);
```

```
// Основной цикл игры, повторяется до тех пор, пока не будет выполнено
// условие GameWon == true.
do
{
   // Проход по всем игрокам для каждого круга игры
   for (currentPlayer = 0; currentPlayer < players.Length;
        currentPlayer++)
   {
      // Вывод информации о текущем игроке, об имеющихся у него на
      // руках картах и о карте, лежащей 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);

      // Вывод подсказки игроку — либо взять открытую карту со стола,
      // либо новую карту из колоды
      bool inputOK = false;
      do
      {
         Console.WriteLine("Press T to take card in play or D to " +
                           "draw:");
         string input = Console.ReadLine();
         if (input.ToLower() == "t")
         {
            // Добавление карты, взятой со стола, к картам,
            // находящимся у игрока
            Console.WriteLine("Drawn: {0}", playCard);
            players[currentPlayer].PlayHand.Add(playCard);
            inputOK = true;
         }
         if (input.ToLower() == "d")
         {
            // Добавление карты, взятой из колоды, к картам,
            // находящимся у игрока
            Card newCard;
            // Добавление карты только в том случае, если она еще
            // не на руках
            bool cardIsInPlayerHand;
            do
            {
               newCard = playDeck.GetCard(currentCard++);
               cardIsInPlayerHand = false;
               // Просмотр в цикле карт всех игроков с целью
               // обнаружения карты newCard у кого-либо на руках
               foreach (Player testPlayer in players)
               {
                  cardIsInPlayerHand |=
                              testPlayer.PlayHand.Contains(newCard);
               }
            } while (cardIsInPlayerHand);

            // Включение обнаруженной карты в карты, находящиеся
            // на руках
            Console.WriteLine("Drawn: {0}", newCard);
            players[currentPlayer].PlayHand.Add(newCard);
            inputOK = true;
         }
      } while (inputOK == false);
```

```
                    // Вывод перенумерованных карт в новом раскладе
                    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 снести какую-либо карту
                    inputOK = false;
                    int choice = -1;
                    do
                    {
                        Console.WriteLine("Choose card to discard:");
                        string input = Console.ReadLine();
                        try
                        {
                            // Попытка преобразовать введенную информацию в допустимый
                            // диапазон карт
                            choice = Convert.ToInt32(input);
                            if ((choice > 0) && (choice <= 8))
                                inputOK = true;
                        }
                        catch
                        {
                            // В случае неудачи при попытке преобразовать введенную
                            // информацию, продолжаем выводить приглашение
                        }
                    } while (inputOK == false);

                    // Помещаем ссылку на сносимую карту в playCard (кладем карту
                    // на стол), после чего удаляем ее из карт, находящихся на руках
                    playCard = players[currentPlayer].PlayHand[choice - 1];
                    players[currentPlayer].PlayHand.RemoveAt(choice - 1);
                    Console.WriteLine("Discarding: {0}", playCard);

                    // Разделение текста, выводящегося для различных игроков
                    Console.WriteLine();

                    // Проверка на предмет того, не выиграл ли кто-либо из игроков,
                    // и выход из цикла по игрокам в таком случае
                    GameWon = players[currentPlayer].HasWon();
                    if (GameWon == true)
                        break;
                }
            } while (GameWon == false);

            // Завершение игры и возврат информации о выигравшем игроке.
            return currentPlayer;
        }
    }
}
```

Вы можете развлечься и поиграть в эту игру, только не забудьте уделить некоторое время подробному изучению кода. Вы также можете попробовать включить точку останова в метод `Reshuffle()` и запустить игру с 7 игроками. Если при этом карты будут постоянно сдаваться, а сданная карта — сбрасываться, то очень скоро произойдет пересдача, поскольку при игре с 7 игроками остаются свободными только три карты. Таким способом вы сможете убедиться, что все работает правильно, наблюдая за тем, как эти три карты появляются заново.

Итоги

В этой главе мы рассмотрели важную тему, посвященную событиям и их обработке. Несмотря на то, что тема эта довольно сложна и поначалу с трудом воспринимается, код, который используется при работе с событиями, оказывается очень простым, и мы обязательно будем широко использовать обработчики событий в остальной части книги.

Рассмотрев несколько простых иллюстративных примеров событий и способов их обработки, мы также внесли соответствующие изменения в библиотеку `CardLib`, которой мы занимались на протяжении нескольких предшествующих глав. После завершения создания библиотеки она была использована в простом приложении для карточной игры. Само приложение может служить демонстрацией большинства приемов программирования, с которыми мы успели познакомиться в первой части настоящей книги.

Этой главой завершается не только исчерпывающее описание ООП, применительно к программированию на C#, но и описание языка C# как такового. Начиная с этого момента мы будем применять полученные знания для реализации более сложных сценариев, например, для создания Windows- и web-приложений, а также для дальнейшего использования .NET Framework.

Упражнения

1. Напишите код, используя синтаксис общего назначения (`object sender, EventArgs e`) для обработки либо события `Timer.Elapsed`, либо события `Connection.MessageArrived`, которые поступают из кода, описанного в данной главе. Обработчик должен выводить строку, указывающую на тип полученного события, а также либо поле `Message` параметра `MessageArrivedEventArgs`, либо свойство `SignalTime` параметра `ElapsedEventArgs` — в зависимости от типа полученного события.

2. Измените приведенный пример карточной игры так, чтобы осуществлялась проверка более интересного условия выигрыша, а именно: игрок считается выигравшим в том случае, если у него на руках оказываются два "набора" карт, один из которых состоит из трех карт, а другой из четырех. В качестве набора карт может рассматриваться либо последовательность карт одинаковой масти (например, тройка, четверка, пятерка и шестерка червей), либо несколько карт одинакового достоинства (например, двойка червей, двойка бубен, двойка пик).

13

Использование управляющих элементов Windows Form

В последние годы Visual Basic получил широкое признание, поскольку он предоставляет программистам удобные инструменты для создания весьма детализированных интерфейсов с помощью программы разработки форм. Это обстоятельство, дополненное очень легким для изучения языком программирования, формирует наилучшую среду для быстрой разработки приложений. В частности, Visual Basic, как и ряд других систем, предназначенных для быстрой разработки приложений, например, Delphi, предоставляет набор управляющих элементов, которые разработчик может использовать, чтобы быстро создать пользовательский интерфейс (user interface, UI) для своего приложения.

В основе большинства Windows-приложений, написанных на Visual Basic, лежит программа разработки форм. Процедура создания пользовательского интерфейса заключается в том, чтобы переносить мышью различные управляющие элементы из панели инструментов в создаваемую форму, размещая их в тех местах, в которых они должны, по вашему мнению, находиться при выполнении программы, а затем с помощью двойного щелчка мышью на конкретном элементе добавить для него обработчик событий. Управляющие элементы, предоставляемые компанией Microsoft, наряду с подобными элементами, создаваемыми на заказ, можно приобрести по вполне разумным ценам. Они предоставляют программисту беспрецедентную возможность многократного использования тщательно выверенного и протестированного кода, которым можно воспользоваться, щелкнув мышью. То, что являлось основополагающим для Visual Basic, теперь, посредством Visual Studio.NET, оказывается доступным и для тех, кто программирует на C#.

Большинство управляющих элементов, использовавшихся до появления .NET, были — и до сих пор остаются — специальными COM-объектами, известными под названием управляющих элементов ActiveX. Эти элементы обычно могут использоваться в процессе как разработки, так и в выполнения приложения. У каждого управляющего элемента имеется некоторый набор свойств, позволяющий программисту выполнять определенные настройки, такие, как задание цвета фона, заголовка и его местоположения в форме. Управляющие элементы, которые мы будем рассматривать в настоящей главе, очень похожи на управляющие элементы ActiveX, однако таковыми не являются — они представляют собой модули .NET. Тем не менее, по-прежнему существует возможность использовать элементы, которые были разработаны для предыдущих версий Visual Studio, хотя при этом и возникает некоторое замедление выполнения кода, поскольку .NET приходится

использовать свою собственную оболочку для управляющих элементов. По вполне очевидным причинам при разработке .NET компания Microsoft не стремилась к избыточному увеличению набора управляющих элементов и предусмотрела возможность использовать уже существующие элементы, несмотря на то, что вновь создаваемые элементы представляют собой .NET-компоненты в чистом виде.

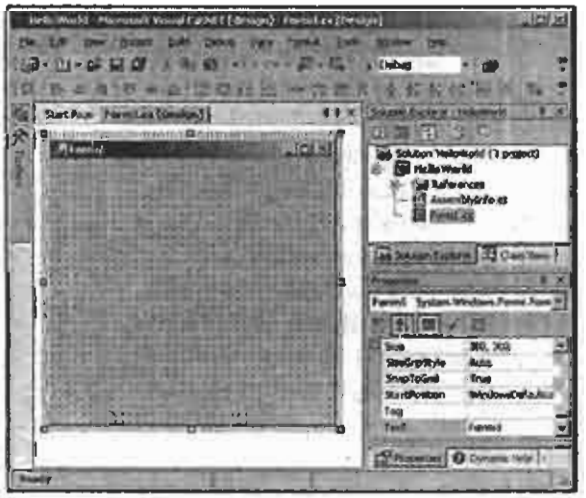
Эти модули .NET могут разрабатываться так, чтобы их можно было использовать в любом языке программирования Visual Studio, и очень хочется надеться, что развивающаяся индустрия создания компонентов проникнется этой идеей и начнет производить чистые .NET-компоненты. Вопрос о том, каким образом можно самостоятельно создавать управляющие элементы, будет рассмотрен в следующей главе.

> *Подробное объяснение того, что представляют собой модули .NET, дается в главе 21. Рекомендуем обратиться к ней, если вы хотите узнать о модулях больше.*

В этой главе мы познакомимся с программой создания форм, особенно с тем, каким образом можно использовать набор управляющих элементов, поставляемый вместе с Visual Studio.NET. Представить все имеющиеся в наличии элементы — задача в рамках настоящей книги неосуществимая, поэтому будут рассмотрены только те из них, которые являются наиболее распространенными, начиная от надписей и текстовых окон и заканчивая средствами просмотра списков и линейками состояний.

Windows Form Designer

Windows Form Designer (программа разработки форм для Windows) — это основное средство, которое используется при создании пользователем собственного интерфейса. Конечно, можно разработать форму и без Visual Studio.NET, но создание интерфейса в Notepad может оказаться весьма болезненной процедурой.



Чтобы познакомимся со средой, которую мы будем использовать, запустите Visual Studio.NET и создайте новый проект C# Windows Application, выбрав в меню **File | New | Project**. В появившемся диалоговом окне щелкните мышью на **Visual C# Projects** в дереве, расположенном с левой стороны, а затем в списке, находящемся справа, выберите **Windows Application**. На данном этапе просто используйте имя, предложенное Visual Studio по умолчанию, и нажмите **OK**. Это должно привести к появлению окна, очень похожего на изображенное на рисунке слева.

Если вы уже знакомы с программой создания форм в Visual Studio, то обратите внимание на сходство — совершенно очевидно, что кто-то решил, что эта программа является победителем и что ее следует использовать и в других языках программирования Visual Studio. Если же вы не знакомы с программой создания форм Visual Studio, то следует пройти последовательно по всем панелям управления, поскольку в этом диалоговом окне происходит не так уж много событий.

В центре экрана расположена форма, разработкой которой вы собираетесь заняться. Есть возможность цеплять мышью и переносить в нее управляющие элементы из панели с инструментами. На рисунке, приведенном выше, панель инструментов находится в свернутом виде, однако если переместить курсор мыши влево до конца экрана на вкладку Toolbox, то она раскроется. Теперь можно щелкнуть мышью на значке, расположенном справа вверху, и панель раскроется. Это приведет к некоторой перекомпоновке рабочего пространства: панель инструментов расположится вверху и не будет налагаться на разрабатываемую форму. Подробнее панель инструментов и то, что в ней содержится, будет рассматриваться ниже.

В свернутом виде на левой линейке прокрутки находится Server Explorer, представленный в виде иконки, изображающей компьютеры, в самом верху вкладки панели инструментов. Можно представить его как сокращенную версию Windows Control Panel. Отсюда осуществляется поиск компьютеров в сети, добавление и удаление соединения с базами данных и многое другое.

В правой части окна располагаются две панели управления. Справа вверху находятся Solution Explorer и окно просмотра классов. В Solution Explorer можно просматривать все открытые проекты и привязанные к ним файлы. Щелкнув мышью на вкладке, расположенной в нижней части Solution Explorer, можно активизировать Class Viewer — программу просмотра классов, позволяющую просматривать все классы, которые используются в проектах, а также все классы, производными которых они являются.

В правой нижней части экрана находится панель Properties (свойства), где содержатся все свойства выбранного элемента, что упрощает их использование и редактирование. Мы будем использовать эту панель в настоящей главе позже.

На этой панели также есть вкладка Dynamic Help (помощь в процессе работы). Здесь будет выводиться краткая вспомогательная информация обо всех выбираемых объектах и даже информация о коде в процессе его набора. Если в вашем компьютере используется устаревший микропроцессор или не очень много оперативной памяти, то мы рекомендуем удалить эту вкладку из панели. Кроме тех случаев, когда она действительно нужна, процедура поиска help-файлов существенно замедляет работу.

Панель инструментов



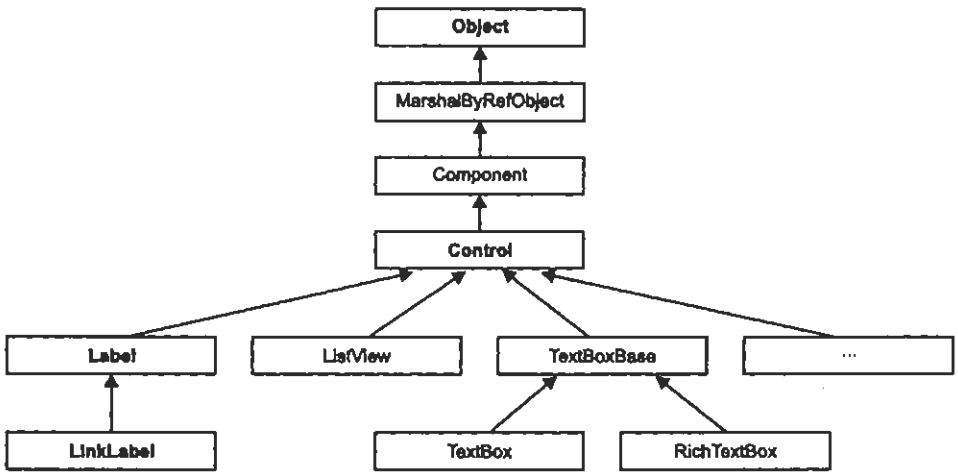
Познакомимся с панелью инструментов поближе. Переместите курсор мыши на панель инструментов в левой части экрана и разверните ее на переднем плане, щелкнув мышью на значке в верхнем правом углу панели.

> *Если вы случайно удалите панель инструментов, нажав вместо этого X, то это можно исправить, либо выбрав пункт Toolbox в меню View, либо нажав* Ctrl-Alt-X.

На панели инструментов содержатся все управляющие элементы, которые доступны разработчику, использующему .NET. В частности, предоставляется возможность выбора элементов, наиболее важных для разработки Windows Application. Если же вы решили создавать проект Web Forms, а не проект Windows Application, то потребуется использовать иную панель инструментов. Вы не ограничены набором, который представлен на панели, и можете изменять его в соответствии со своими нуждами. Сейчас же мы сосредоточимся на элементах, которые изначально находятся в панели инструментов, изображенной на приведенном слева рисунке.

Управляющие элементы

Большинство управляющих элементов в .NET являются производными от класса `System.Windows.Form.Control`. В нем определяются базовые функциональные возможностн управляющих элементов, н по этой причине многие свойства и события являются идентичными. Многие из этих классов сами по себе являются базовыми для других управляющих элементов, как в случае с классами `Label` и `TextBoxBase`, изображенными на следующей диаграмме:



Некоторые управляющие элементы, которые называются пользовательскими, являются производными от класса `System.Windows.Form.UserControl`. Этот класс, в свою очередь, является производным от класса `Control` и предоставляет все функциональные возможности, которые необходимы для самостоятельного создания управляющих элементов. Этот класс будет рассматриваться в главе 14. Элементы, применяемые при разработке пользовательских интерфейсов в web, являются производными класса `System.Web.UI.Control`.

Свойства

Все управляющие элементы обладают набором свойств, которые используются для управления поведением этих элемеитов. Класс `Control`, являющийся базовым для большинства управляющих элементов, обладает свойствами, которые другие элементы либо непосредственно иаследуют, либо переопределяют для достижения требуемого образа действий.

В следующей ниже таблице приведены некоторые из наиболее распространенных свойств класса `Control`. Эти свойства будут присутствовать в большинстве управляющих элементов, с которыми мы познакомимся в этой главе, и, таким образом, нам не придется подробно останавливаться на их описании по второму разу, за исключением тех случаев, когда для рассматриваемого управляющего элемента поведение какого-либо свойства является отличным. Эта таблица не претендует на то, чтобы быть исчерпывающей; если вы хотите познакомиться с полным перечнем свойств данного класса, то вам следует обратиться в библиотеку MSDN.

Имя	Режим доступа	Описание
`Anchor`	Чтение/запись	С помощью этого свойства можно определять, каким образом будет вести себя управляющий элемент при изменении размеров своего контейнера. Более подробно это свойство будет объясиено инже.
`BackColor`	Чтение/запись	Определяет цвет фона управляющего элемента.
`Bottom`	Чтение/запись	Задавая значение этого свойства, можно определнть расстояние от верха окна до нижней частн управляющего элемеита. Это не то же самое, что заданне высоты управляющего элемеита.
`Dock`	Чтенне/запись	Позволяет делать управляющий элемент прилегающим к краям окна. Более подробно это свойство будет объяснено ннже.
`Enabled`	Чтение/запись	Присванванне свойству `Enabled` значения `true` обычно означает, что данный управляющнй элемент в состоянии прнннмать ннформацню, вводящуюся пользователем. Прнсваивание свойству `Enabled` значения `false` обычно означает, что даиный управляющий элемент делать этого не в состоянии.
`ForeColor`	Чтение/запись	Определяет цвет управляющего элемента.
`Height`	Чтение/запись	Определяет расстояние от верхиего до нижиего края управляющего элемента.
`Left`	Чтенне/запись	Определяет положение левого края управляющего элемента относительно левого края окна.
`Name`	Чтенне/запись	Имя управляющего элемента. По этому имеин к управляющему элементу можно обращаться программным путем.
`Parent`	Чтение/запись	Управляющнй элемент, от которого происходит даниый управляющий элемент.
`Right`	Чтение/запись	Определяет положение правого края управляющего элемента относнтельно правого края окиа.
`TabIndex`	Чтение/запись	Определяет порядковый номер управляющего элемента для данного коитейнера.
`TabStop`	Чтение/запись	Определяет, возможно ли осуществлять доступ к данному управляющему элемеиту с помощью клавиши *Tab*.
`Tag`	Чтенне/запись	Это свойство обычно не нспользуется самим управляющнм элементом; оно включеио в эту таблицу, поскольку в ием может хранntься информация о самом управляющем элементе. Если значение этого свойства присваивается посредством программы разработки Windows Form, то допустимым является только текстовая строка.
`Top`	Чтение/запись	Определяет положение верхнего края управляющего элемента относительно верхиего края окиа.
`Visible`	Чтение/запись	Определяет, является ли управляющий элемент видимым в процессе выполнения.
`Width`	Чтение/запись	Определяет ширину управляющего элемента.

Свойства Anchor и Dock

Эти два свойства оказываются особенно полезнымн при разработке формы. Задача гарантировать, что окно не примет беспорядочный вид в том случае, если пользователь изменит его размеры, оказывается далеко не трнвиальной, и для ее решения было написано бесчисленное количество строк кода. Множество программ решают эту проблему, налагая запрет на нзмененне размера окна, что, безусловно, является нанболее простым, но не самым лучшим способом справиться с этой задачей. Свойства `Anchor` и `Dock`, которые появнлись вместе с .NET, позволяют решнть эту проблему, не иаписав ии единой строчки кода.

Свойство Anchor позволяет определить, каким образом должен вести себя управляющий элемент в тот момент, когда пользователь изменяет размер окна. Можно указать, что элемент должен изменить свои размеры, сохранив пропорции внешних размеров, или оставить размеры неизменнымн, сохранив свою позицию относнтельно границ окна.

Свойство Dock связано со свойством Anchor. Оно может быть использовано для того, что бы управляющий элемент прилегал к краю своего контейнера. В том случае, если пользователь изменит размеры окна, элемент все равно останется прилегающим к его краю. Если, например, задано, что управляющий элемент должен прилегать к нижней границе окна, то он будет всегда изменять свое местоположение так, чтобы занимать иижнюю часть экрана независимо от размеров окна. Размер же управляющего элемента изменяться при этом не будет — он всегда будет прилегать к соответствующему краю окна.

Для того чтобы правильно использовать свойство Anchor, внимательно изучите пример текстового окна, приведенный ниже в этой главе.

События

Когда пользователь нажимает на кнопку или щелкает по ней мышью, то вам как программисту-разработчику данного приложения, хотелось бы получить информацию о том, что это произошло. С этой целью управляющие элементы используют **события**. Класс Control определяет несколько событий, являющихся общнмн для всех элементов, которые будут использоваться в настоящей главе. Эти события описываются в приведенной ниже таблице, которая представляет собой выборку наяболее распространенных событий; полный список можно найти в библиотеке MSDN:

Имя	Описание
Click	Наступает в том случае, когда происходит щелчок мышью на управляющем элементе. Это событне наступает также и в том случае, если пользователь нажимает клавишу *Enter*.
DoubleClick	Наступает в том случае, когда иа управляющем элементе происходит двукратный шелчок мышью. Использование события Click для некоторых управляющих элементов, таких как управляющий элемент Button, означает, что событне DoubleClick никогда ие будет вызываться.
DragDrop	Наступает, когда завершается процедура переноса мышью, другими словами, в тот момент, когда объект перенесен мышью иа управляющий элемент и пользователь отпускает кнопку мыши.
DragEnter	Возникает в тот момент, когда переносимый мышью объект оказывается внутри границ управляющего элемента.
DragLeave	Возникает в тот момент, когда переносимый мышью объект оказывается вне границ управляющего элемента.
DragOver	Возникает в тот момент, когда переносимый мышью объект накрывает управляющий элемент.
KeyDown	Наступает в момент нажатия клавишн в том случае, еслн управляющий элемент находится в фокусе. Это событие всегда предшествует событиям KeyPress и KeyUp.
KeyPress	Наступает в момент нажатия клавиши в том случае, еслн управляющий элемент находится в фокусе. Это событне всегда возникает после события KeyDown и перед событнем KeyUp. Разница между KeyDown и KeyPress заключается в том, KeyDown передает код нажатой клавишн, в то время как KeyPress передает значение типа char, соответствующее нажатой клавише.

Продолжение таблицы

Имя	Описание
`KeyUp`	Наступает в тот момент, когда клавиша отпускается, в том случае если управляющий элемент находится в фокусе. Это событие всегда возникает после событий `KeyDown` и `KeyPress`.
`GotFocus`	Возникает в тот момент, когда управляющий элемент оказывается в фокусе. Не рекомендуется использовать это событие для выполнения проверки допустимости управляющих элементов. Для этих целей следует использовать события `Validating` и `Validated`.
`LostFocus`	Возникает в тот момент, когда управляющий элемент оказывается не в фокусе. Не рекомендуется использовать это событие для выполнения проверки допустимости управляющих элементов. Для этих целей следует использовать события `Validating` и `Validated`.
`MouseDown`	Возникает в тот момент, когда курсор мыши находится на управляющем элементе, а кнопка мыши находится в нажатом состоянии. Это не то же самое, что событие `Click`, поскольку событие `MouseDown` возникает, когда кнопка мыши нажата и до того момента, когда она будет отпущена.
`MoseMove`	Возникает постоянно по мере того, как курсор мыши передвигается по управляющему элементу.
`MouseUp`	Возникает в тот момент, когда курсор мыши находится на управляющем элементе, а кнопка мыши отпущена.
`Paint`	Возникает в момент, когда управляющий элемент рисуется на экране.
`Validated`	Это событие генерируется, когда управляющий элемент, свойство которого `CausesValidation` (требует проверки допустимости) равно `true`, должен оказаться в фокусе. Оно наступает, после того как завершается событие `Validating`, и свидетельствует о том, что проверка допустимости завершена.
`Validating`	Это событие генерируется, когда управляющий элемент, свойство которого `CausesValidation` равно `true`, должен оказаться в фокусе. Обратите внимание на то, что управляющий элемент, для которого будет выполняться проверка допустимости, — это элемент, теряющий фокус, а не получающий его.

Большинство этих событий встретятся в примерах, которые будут приводиться в настоящей главе.

Теперь следует перейти к рассмотрению собственно управляющих элементов, и начнем мы с элемента, который уже встречался в предыдущих главах, — с элемента `Button` (кнопка).

Управляющий элемент Button

В Visual Studio.NET есть три типа кнопок. Переключатели и окна с выбором также представляют собой кнопки. По этой причине класс `Button` не является производным непосредственно от класса `Control`, он является производным от класса `ButtonBase`, который, в свою очередь, является производным от класса `Control`. В данном разделе мы сосредоточим внимание на элементе `Button`, а к переключателям и окнам для выбора вернемся ниже в настоящей главе.

Управляющий элемент в виде кнопки присутствует практически во всех диалогах Windows. Кнопки используются прежде всего для выполнения задач следующих трех видов:

- Для завершения диалога с сохранением состояния (например, кнопки OK и Cancel).

- Для выполнения каких-либо действий над данными, введенными в процессе диалога (например, при нажатии кнопки Search (поиск) после ввода некоторого критерия поиска).
- Для открытия другого диалога или приложения (например, кнопка Help (помощь)).

Работать со управляющими элементами, представляющими собой кнопки, оказывается очень просто. Вся работа заключается в добавлении элемента к форме и двойном щелчке мышью на нем для ввода кода, обрабатывающего событие `Click`,— этого оказывается вполне достаточно для большинства приложений, с которыми вы будете работать.

Рассмотрение некоторых наиболее часто использующихся свойств и событий данного элемента поможет получить представление о задачах, которые могут быть решены с его помощью. Потом мы создадим небольшой пример, который позволит продемонстрировать некоторые основные свойства и события, используемые при работе с кнопками.

Свойства управляющего элемента Button

Свойства в качестве членов самой кнопки будут перечисляться даже в том случае, если технически они описываются в базовом классе `ButtonBase`. Ниже приводится описание только наиболее часто употребляемых свойств. Полный список можно найти в MSDN:

Имя	Режим доступа	Описание
`FlatStyle`	Чтение/запись	Позволяет изменять стиль кнопки. Если задать стиль `PopUp`, то кнопка будет выглядеть плоской до тех пор, пока пользователь не наведет на нее курсор мыши. В этот момент кнопка примет свой обычный объемный вид.
`Enabled`	Чтение/запись	Описание этого свойства приводится здесь, даже несмотря на то, что оно является производным от класса `Control`, поскольку это очень важное свойство для работы с кнопками. Если присвоить свойству `Enabled` значение `false`, то это будет означать, что изображение кнопки становится более темным и при нажатии на нее ничего не происходит.
`Image`	Чтение/запись	Позволяет задать рисунок (в двоичном виде, в виде иконки и т. д), который будет изображаться на кнопке.
`ImageAlign`	Чтение/запись	Позволяет определять местоположение рисунка, размещаемого на кнопке.

События управляющего элемента Button

Наиболее часто используемым событием для кнопок является событие `Click`. Это событие наступает, когда пользователь щелкает мышью на кнопке, что означает, что он нажимает левую кнопку мыши и отпускает ее, пока курсор мыши находится на данном управляющем элементе. Отсюда, в частности, следует, что в том случае, если сначала будет нажата левая кнопка мыши, а затем курсор мыши будет уведен со управляющего элемента до того, как она будет отпущена, то это не приведет к наступлению события `Click`. Кроме того, событие `Click` возникает, если данная кнопка находится в фокусе, а пользователь нажимает клавишу *Enter*.

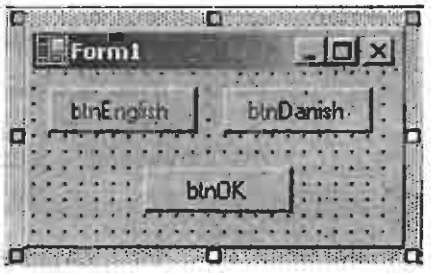
Если в вашей форме используется какая-либо кнопка, то вы должны непременно обрабатывать это событие.

В примере будет создан диалог, в котором будут использоваться три кнопки. Первые две кнопки будут изменять используемый язык с английского на датский и обратно (можно использовать любой другой язык по своему усмотрению). Последняя кнопка будет использоваться для завершения диалога.

Практикум: тестирование управляющего элемента Button

1. Откройте Visual Studio.NET и создайте новое приложение C# Windows Application. Назовите это приложение `ButtonTest`.

2. Разверните Toolbox и выполните три двукратных щелчка мышью на управляющем элементе `Button`. Затем передвиньте кнопки и установите соответствующий размер формы, как показано на рисунке справа.



3. Щелкните правой кнопкой мыши на **Properties**. Затем измените свойство `Name` для всех трех кнопок, как показано на рисунке, выбрав поле редактирования свойства `Name` на панели управления **Properties** и набрав необходимый текст.

4. Измените свойство `Text` каждой из трех кнопок на соответствующий текст за исключением первых трех букв (btn).

5. Перед текстом необходимо вывести соответствующий флажок, чтобы было понятно, о каком языке идет речь. Выберите кнопку **English** и найдите свойство `Image`. Щелкните мышью на (...) справа от него, для того чтобы перейти в диалоговое окно, в котором можно выбирать рисунки. Иконки с различными флагами поставляются вместе с Visual Studio.NET. Если вы инсталлировали Visual Studio.NET на стандартное место (имеется в виду англоязычная версия), то они должны располагаться в директории `C:\Program Files\Microsoft Visual Studio.NET\Common7\Graphics\icons\Flags`. Выберите иконку `flguk.ico`. Повторите ту же самую процедуру для кнопки **Danish**, выбрав на этот раз файл `flgden.ico` (по желанию можно выбрать любой другой флаг из имеющихся в данной директории).

6. На данный момент текст и иконка, расположенные на кнопке, накладываются друг на друга, поэтому необходимо изменить местоположение иконки. Для обеих кнопок — **English** и **Danish** — измените значение свойства `ImageAlign` на `MiddleLeft`.

7. В этот момент можно скорректировать ширину кнопок, с тем чтобы текст не начинался непосредственно в том месте, где заканчивается изображение. Для этого последовательно выберите каждую кнопку и растяните ее правый край.

8. В заключение щелкните мышью на форме и измените свойство `Text` на "Do you speak English?" (Говорите ли вы по-английски?).



Вот и все, что потребовалось для создания пользовательского интерфейса нашего диалога. Должно получиться нечто, выглядящее приблизительно так, как показано на рисунке слева.

Теперь можно добавить в диалоговое окно обработчики событий. Щелкните мышью два раза на кнопке English — вы попадете непосредственно на обработчик события. `Click` является событием, используемым для кнопок по умолчанию; это именно то событие, которое создается при двойном щелчке мышью на кнопке. У других управляющих элементов по умолчанию используются другие события.

Добавление обработчиков событий

При двойном щелчке мышью на управляющем элементе в коде, который лежит в основе данной формы, происходят две вещи. Во-первых, в методе `InitializeComponent()` создается подписка на событие:

```
this.btnEnglish.Click += new System.EventHandler(this.btnEnglish_Click);
```

Если возникает необходимость подписаться на событие, отличное от события по умолчанию, то придется заносить соответствующий код самостоятельно, чем мы и будем заниматься на протяжении всей оставшейся главы. Представляется важным помнить, что код метода `InitializeComponent()` перезаписывается каждый раз, когда вы переключаетесь из режима разработки и переходите к коду. По этой причине никогда не выполняйте подписку в этом методе. Вместо этого следует использовать конструктор данного класса.

Второе, что происходит,— это добавляется собственно обработчик события:

```
private void btnEnglish_Click(object sender, System.EventArgs e)
{
    this.Text = "Do you speak English?";
}
```

Имя метода образуется посредством объединения имени управляющего элемента, символа подчеркивания и имени обрабатываемого события. Первый параметр — `object sender` — содержит выбранный управляющий элемент. В данном примере, это всегда будет элемент, который входит в имя метода, хотя в некоторых других случаях несколько управляющих элементов могут использовать один и тот же метод для обработки события, и в таком случае по значению этого параметра можно точно определить, какой именно элемент вызывает данный метод. В примере с текстовым окном, приведенном в этой главе ниже, демонстрируется, каким образом можно использовать один и тот же метод одновременно для нескольких управляющих элементов. Во втором параметре — `System.EventArgs e` — содержится информация о том, что именно произошло. В данном случае нет необходимости использовать информацию, хранящуюся в обоих параметрах.

Ключевое слово `this` обозначает текущий экземпляр данного класса. Поскольку класс, с которым мы работаем в настоящий момент, представлен именно таким экземпляром, можно получать доступ к его свойствам и управляющим элементам с помощью этого слова. Задавая значение свойства `Text`, используемого с ключевым словом `this`, как это делается в приведенном выше коде, мы, на самом деле, задаем значение свойства `Text` текущего экземпляра формы.

Вернитесь в Form Designer и щелкните мышью два раза на кнопке Danish для перехода к обработчику событий для этой кнопки. Вот его код:

```
public void btnDanish_Click(object sender, System.EventArgs e)
{
this.Text = "Taler du Dansk?";
}
```

Этот метод идентичен методу `btnEnglish_Click` за исключением текста на датском языке. Обработчик событий для кнопки OK добавляется точно таким же способом, как и два предыдущих. Однако код в этом случае будет немного отличаться:

```
public void btnOK_Click(object sender, System.EventArgs e)
{
Application.Exit();
}
```

С помощью этого метода завершается выполнение приложения. Откомпилируйте его, запустите и понажимайте на кнопки. Картина будет похожей на рисунок справа.

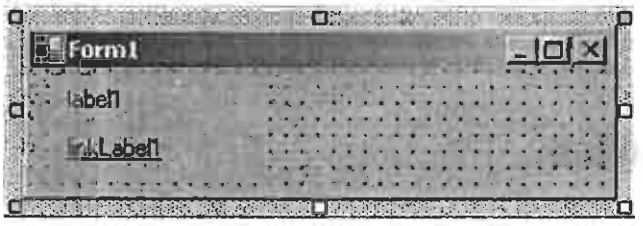


Управляющие элементы Label и LinkLabel

Управляющий элемент `Label` является наиболее употребляемым. Он есть практически во всех приложениях Windows. `Label` — простой элемент, предназначенный для единственной цели: он представляет собой надпись или короткую рекомендацию, объясняющую что-либо пользователю данной формы.

Стандартно Visual Studio.NET поставляется с двумя разновидностями надписей, представленными двумя различными способами:

- `Label` — стандартная надпись в Windows
- `LinkLabel` — надпись, похожая на стандартную (и являющаяся производной от нее), но представленная в виде интериет-ссылки (гиперссылки)



Эти два элемента находятся в верхней части панели управления вкладки Windows Form. На рисунке слева показана форма, в которую были перенесены оба элемента типа `Label`, с тем чтобы продемонстрировать внешние отличия между названными элементами.

Если вы программировали на Visual Basic, то, вероятно, обращали внимание, что для задания выводимого текста используется свойство `Text`, а не `Caption`. Далее вы узнаете, что во всех присущих .NET управляющих элементах для задания основного текста используется именно свойство `Text`. До появления .NET свойства `Caption` и `Text` были взаимозаменяемыми.

Обычно для стандартного элемента Label не требуется никакого кода в обработчике событий. Однако для LinkLabel некоторый дополнительный код необходим, если вы хотите предоставить пользователю возможность переходить на указанную в тексте web-страницу.

Управляющий элемент Label обладает на удивление большим количеством свойств, которые можно задавать. Большинство из них являются производными от класса Control, но есть несколько новых. В следующей таблице приведены наиболее распространенные их них. Если не утверждается противное, эти свойства существуют у обоих элементов — Label и LinkLabel.

Имя	Режим доступа	Описание
BorderStyle	Чтение/запись	Позволяет определять стиль обрамления вокруг Label. По умолчанию обрамление отсутствует.
DisabledLinkColor	Чтение/запись	(Только для LinkLabel.) Определяет цвет, принимаемый LinkLabel после нажатия на него пользователем.
FlatStyle	Чтение/запись	Определяет внешнее представление управляющего элемента. Если присвоить этому свойству значение PopUp, то управляющий элемент будет изображаться плоским до тех пор, пока пользователь не наведет на него курсор мыши. После этого он несколько приподнимется.
Image	Чтение/запись	Это свойство позволяет задавать единственный рисунок (двоичный, иконку и т. д.), выводимый на тексте.
ImageAlign	Чтение/запись	Определяет место, где в Label будет располагаться рисунок, заданный в свойстве Image.
LinkArea	Чтение/запись	(Только для LinkLabel.) Позволяет задавать часть текста, которая будет выводиться в качестве ссылки.
LinkColor	Чтение/запись	(Только для LinkLabel.) Определяет цвет ссылки.
Links	Только чтение	(Только для LinkLabel.) Управляющий элемент LinkLabel может содержать более одной ссылки. Данное свойство позволяет находить требуемую ссылку. Этот управляющий элемент позволяет отслеживать ссылки, выведенные в тексте.
LinkVisited	Только чтение	(Только для LinkLabel.) Возвращает значение, которое указывает, посещалась данная ссылка или нет.
Text	Чтение/запись	Позволяет задавать выводимый текст.
TextAlign	Чтение/запись	Определяет, каким образом текст располагается в управляющем элементе.

Управляющий элемент Textbox

Управляющие элементы типа Textbox (текстовое окно) следует использовать в тех случаях, когда пользователь вводит текст, о котором ничего не известно на момент разработки (например имя пользователя). Основная задача текстовых окон — позволить пользователю вводить текст, но, поскольку допускается ввод произвольных символов, можно заставить пользователя вводить, например только цифры.

Стандартно .NET поставляется с двумя основными управляющими элементами — Textbox и RichTextbox (RichTextbox будет обсуждаться в данной главе ниже). Оба элемента являются производными от базового класса TextBoxBase, который, в свою очередь, является производным от класса Control.

Класс `TextBoxBase` обеспечивает базовые функциональные возможности по работе с текстом в текстовом окне, такие как выбор текста, вырезание и вставка текста, а также широкий спектр событий. Рассмотрим более простой элемент — `Textbox`. Мы создадим пример, демонстрирующий осиовные свойства `Textbox`, и на его основе продемонстрируем управляющий элемент `RichTextbox`.

Свойства управляющего элемента Textbox

У данного элемента имеется очень много свойств, поэтому в следующий перечень включены только наиболее распространенные из них.

Имя	Режим доступа	Описание
`CausesValidation`	Чтение/запись	Когда элемент, у которого данному свойству присвоено значение `true`, должен оказаться в фокусе, то генерируются два события — `Validating` и `Validated`. Их можно обработать с целью проверки допустимости данных, введенных в управляющий элемент, который выходит из фокуса. Это может привести к тому, что этот элемент никогда не окажется в фокусе. Соответствующие события будут обсуждаться ниже.
`CharacterCasing`	Чтение/запись	Позволяет определнть, производится ли изменение регистра вводимых снмволов. Это свойство может нметь одно из трех значений: • `Lower` — весь вводимый в окно текст переводится в нижний регистр. • `Normal` — текст остается без изменений. • `Upper` — весь вводимый в окно текст переводится в верхний регистр.
`MaxLength`	Чтение/запись	Значение, которое определяет максимальную длину текста в `Textbox`, выраженную в символах. Если длина ограничена только объемом имеющейся в наличин памяти, данному свойству следует присвоить значение, равное нулю.
`Multiline`	Чтение/запись	Указывает, допускает лн даниый управляющий элемент наличие нескольких строк. Многострочный элемент позволяет одновременный вывод нескольких строк текста.
`PasswordChar`	Чтение/запись	Указывает, что текст, находящийся в однострочном окие, должен заменяться на "секретные" символы. Если свойство `Multiline` имеет значение, равное `true`, то это свойство игнорируется.
`ReadOnly`	Чтение/запись	Логическое значенне, указывающее на то, что текст может использоваться в режиме "только чтение".
`ScrollBars`	Чтение/запись	Позволяет указать, что состоящее из нескольких строк окно должно выводиться с линейками прокрутки.
`SelectedText`	Чтение/запись	Определяет выбранный в окне текст.
`SelectionLength`	Чтение/запись	Определяет число символов в выбранном тексте. Если присвоенное этому свойству значение превышает колнчество символов в тексте, оно будет изменено иа разность между количеством снмволов в тексте и значением свойства `SelectionStart`.
`SelectionStart`	Чтение/запись	Определяет начало выделенного текста в текстовом окне.
`WordWrap`	Чтение/запнсь	Определяет, необходимо ли в многострочном окие автоматнчески сворачивать слова в том случае, если длина строки превышает ширину управляющего элемента.

События управляющего элемента Textbox

Тщательная проверка допустимости текста в управляющих элементах типа `Textbox` позволяет превратить обозленных пользователей в довольных.

Вас, вероятно, не раз раздражала ситуация, когда в процессе диалога содержимое окон проверяется на допустимость только после того, как вы нажимаете OK. При таком подходе вы обычно получаете окно с сообщением о том, что данные, введенные в "Textbox номер три", некорректны. После этого вы будете нажимать кнопку OK до тех пор, пока все данные не будут исправлены. Очевидно, что это не самый лучший способ проверки допустимости данных. Итак, что же можно предложить взамен?

Ответ заключается в обработке событий, связанных с проверкой допустимости, которые предоставляются управляющим элементом `Textbox`. Если требуется гарантированно не допускать ввода неверных символов или если нужно, чтобы вводимое в текстовое окно значение находилось в допустимом диапазоне, то необходимо указать пользователю этого управляющего элемента, является ли введенное значение допустимым или нет.

Управляющий элемент `Textbox` предоставляет следующие события (все они наследуются от класса `Control`):

Имя	Описание
`Enter` `GotFocus` `Leave` `Validating` `Validated` `LostFocus`	Первые шесть событий, приведенных в таблице наступают в той последовательности, в которой они перечислены. Они известны под названием "событий, связанных с фокусом" и генерируются при любом изменении фокуса управляющего элемента за исключением двух случаев. События `Validating` и `Validated` происходят только в том случае, если свойство `CausesValidation` имеет значение `true`. Причина, по которой именно управляющий элемент, получающий фокус, исполняет это событие, заключается в том, что могут возникать ситуации, при которых вообще не требуется проводить проверку допустимости, несмотря на изменение фокуса. Пример — нажатие кнопки Help.
`KeyDown` `KeyPress` `KeyUp`	Следующие три события известны как "события, связанные с клавишами". Они позволяют отслеживать и изменять то, что в вводится в управляющий элемент. События `KeyDown` и `KeyUp` получают код, соответствующий нажатой клавише. Это позволяет определять нажатие специальных клавиш, например, *Shift* или *Ctrl* вместе с *F1*. `KeyPress` позволяет получать символ, соответствующий нажатой клавише. Это означает, что символ "a" отличен от символа "A". Это событие оказывается полезным, если требуется задать множество допустимых символов, например, разрешить ввод только численных значений.
`Change`	Наступает при любых изменениях текста в текстовом окне независимо от их характера.

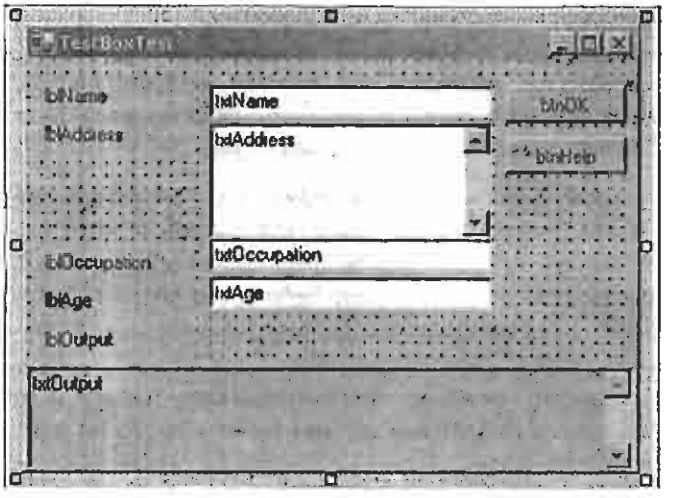
Практикум: тестирование управляющего элемента Textbox

Мы создадим диалог, который позволит вам ввести имя, адрес, род занятий и возраст. Основная задача диалога — научить основам работы со свойствами и событиями, а вовсе не создать какую-то чрезвычайно полезную программу.

Начнем с создания интерфейса пользователя:

1. Выберите пункт меню Files | New и создайте новое Windows Application в разделе C# Projects. Назовите проект `TextBoxTest`.

2. Создайте форму, изображенную на приведенном ниже рисунке, путем переноса мышью надписей, окон и кнопок на рабочую поверхность.



До того как у вас появится возможность переопределить размер двух текстовых окон **txtAddress** и **txtOutput**, необходимо приевоить их свойству `Multiline` значение `true`. Достигнуть этого можно, щелкнув правой кнопкой мыши на соответствующем управляющем элементе и выбрав **Properties** (см. рис. слева).

3. Присвойтс управляющим элсмснтам имена, как показано на рисунке.
4. Присвойте свойству `Text` каждого элемента типа `Textbox` в качестве значения пустую строку, что означает, что в них не будет ничего содержаться в момент первого запуска приложения.
5. Присвойте свойству `Text` всех остальных элементов текст, соответствующий их имени за исключением первых трех символов. Присвойте в качестве значения свойству `Text` формы заголовок, изображениый на рисунке.
6. Присвойте свойству `Scrollbars` двух элементов — `txtAddress` и `txtOutput` — значение `Vertical`.
7. Присвойте свойству `ReadOnly` элемента `txtOutput` значение `true`.
8. Присвойте свойству `CausesValidation` кнопки **btnHelp** значение `false`. Значение этого свойства, равное `false`, позволяет при нажатии пользователем этой кнопки избежать преобразований, связанных с вводом неверных данных.
9. После того как этой форме придан размер, позволяющнй разместить все элементы, следует определить соответствующее поведение этих элементов при изменении размеров формы. Присвойте свойству `Anchor` значения в соответствии с указанными в следующей таблице:

Управляющий элемент	Значение свойства Anchor
Все управляющие элементы типа `Label`	`Top, Bottom, Left`
Все управляющие элементы типа `Textbox`	`Top, Bottom, Left, Right`
Обе кнопкн	`Top, Right`

Подсказка: Можно взять текст свойства `Anchor` из одного управляющего элемента и вставить его в другие элементы, которым требуются те же значения. При этом нет необходимости использовать кнопку, раскрывающую окно.

Причина, по которой управляющий элемент `txtOutput` не описывается как прилегающий к нижней границе формы, заключается в том, чтобы область, предназначенная для вывода текста, должна изменять размеры при изменении размеров формы. Если описать ее как прилегающую к нижней границе, то она всегда бы оставалась там и ее размеры оставались бы неизмеиными.

10. В заключение необходимо учесть еще один момент. Найдите свойства формы `Size` и `MinSize`. От нашей формы будет мало проку, если ее размеры окажутся меньше, чем те, которыми она обладает в настоящий момент, следовательно, свойству `MinSize` следует присвоить то же значение, которое имеет свойство `Size`.

Работа, связанная с определением видимой части формы, на этом завершена. Если запустить приложение в таком виде, то при вводе какого-либо текста или нажатия на какие-либо кнопки ничего не будет происходить, однако если изменить диалоговое окно, то управляющие элементы будут вести себя таким образом, к которому вы стремились, определяя пользовательский интерфейс,— оставаясь на своих местах и изменяя размеры так, чтобы заполнять все окно. Если вы когда-нибудь попытаетесь реализовать ту же задачу на языке программирования вроде Visual Basic 6, то вам станет ясно, от какого гигантского объема работы вы были освобождены.

Теперь настало время перейти к рассмотрению кода. Щелкните правой кнопкой мыши на форме и выберите пункт View Code. Если панель инструментов закреплена, освободите ее, чтобы появилось достаточно пространства для окна ввода кода.

В редакторе находится удивительно малое количество кода. В верхней части описания класса находятся описания управляющих элементов, однако, до тех пор пока вы не развернете область под названием Windows Form Designer Generated Code (код, сгенерированный программой Windows Form Designer), вы не увидите всей проведенной работы. Следует особо подчеркнуть, что не нужно редактировать код, находящийся в этом разделе! После внесения изменений в программу Windows Form Designer будут все исправления будут затерты, или — что еще хуже — в процессе редактирования могут быть внесены такие изменения, в результате которых программа окажется не в состоянии выводить форму.

Представляется полезным потратить некоторое время на то, чтобы просмотреть операторы, расположенные в этом разделе. Это позволит узнать, каким именно образом можно создать Windows-приложение без использования Visual Studio.NET. Все, что находится в данном разделе, может быть введено с помощью Notepad или какого-либо аналогичного текстового редактора и откомпилировано, но такой подход не представляется разумным. Уследить за всем, что происходит в этом разделе, практически невозможно; существует высокая вероятность допустить множество ошибок, а, поскольку вы лишены возможности наблюдать эффект от собственных действий, то надлежащее размещение управляющих элементов становится весьма обременительной задачей. Такая ситуация, однако, предоставляет возможность сторонним производителям программного обеспечения создавать свои собственные варианты среды программирования, конкурирующие с Visual Studio.NET, поскольку компиляторы, используемые для создания форм, включаются в состав .NET Framework, а не в Visual Studio.NET.

Добавление обработчиков событий

Перед тем как приступить к добавлению собственноручно написанного кода, можно извлечь из программы-разработчика последнюю услугу. Вернитесь обратно в Form Designer (щелкнув мышью на вкладке, расположенной в верхней части текстового редактора) и дважды щелкните на кнопке btnOK. Повторите это и для другой кнопки. Подобные действия приводят к созданию обработчика события "щелчок мышью" для каждой из кнопок. В момент, когда мышь щелкает на кнопке OK, нам необходимо передать текст, введенный в текстовые окна, в окно для вывода текста, которое может использоваться в режиме "только чтение".

Ниже приводится код, предназначенный для обработки этих двух событий:

```
private void btnOK_Click(object sender, System.EventArgs e)
{
   // Никаких проверок на предмет недопустимости введенных значений
   // не производится, поскольку в этом нет необходимости

   string output;

   // Конкатенация текстовых значений из всех четырех текстовых окон
   output = "Name: " + this.txtName.Text + "\r\n";
   output += "Address: " + this.txtAddress.Text + "\r\n";
   output += "Occupation: " + this.txtOccupation.Text + "\r\n";
   output += "Age: " + this.txtAge.Text;

   // Вставка нового текста
   this.txtOutput.Text = output;
}

private void btnHelp_Click(object sender, System.EventArgs e)
{
   // Вывод краткого описания каждого текстового окна
   string output;

   output = "Name = Your name + \r\n";
   output += "Address = Your address + \r\n";
   output += "Occupation = Only allowed value is 'Programmer' or empty + "\r\n";
   output += "Age = Your age";

   // Вставка нового текста
   this.txtOutput.Text = output;
}
```

В обеих функциях используется свойство `Text` соответствующего текстового окна, значение которого либо — как в функции `btnOK_Click()` — извлекается или задается, либо — как в функции `btnHelp_Click()` — просто задается.

Мы используем информацию, введенную пользователем, не утруждая себя осуществлением каких-либо проверок на предмет ее корректности. Это означает, что необходимо выполнять соответствующие проверки в каком-нибудь другом месте. Для данного примера существует определенный перечень критериев, которым должен удовлетворять вводимый текст, для того чтобы можно было получить корректные значения:

- Поле с именем пользователя не может оставаться пустым
- Возраст пользователя должен представлять собой число, большее или равное нулю
- Род занятий пользователя должен описываться как Programmer (программист) или оставаться пустым
- Поле с адресом пользователя не может оставаться пустым

Из этого набора условий становится понятно, что для двух текстовых окон (`txtName` и `txtAddress`) необходимо выполнение одних и тех же проверок. Необходимо также предусмотреть исключение возможности ввода пользователем какой-либо некорректной информации в окно `Age`, и, наконец, мы должны посмотреть, является ли пользователь программистом.

Для того чтобы сделать невозможным нажатие кнопки OK до введения всей необходимой информации, следует начать с присвоения свойству `Enabled` кнопки OK

значения false. Это позволит гарантировать, что значение свойства не будет задано до тех пор, пока не будет выполнен код, сгенерированный в InitializeComponent():

```
public Form1()
{
   //
   // Используется для поддержки Windows Form Designer
   //
   InitializeComponent();
   this.btnOK.Enabled = false;
}
```

Теперь нужно приступить к созданию обработчика для тех двух окон, которые необходимо проверять на предмет того, не являются ли они пустыми. Этого можно достичь, подписавшись на событие Validating текстовых окон. Управляющему элементу передается информация о том, что событие должно обрабатываться с помощью метода с именем txtBoxEmpty_Validating().

Также необходим способ, позволяющий получать информацию относительно состояния управляющих элементов. Для этой цели используется свойство текстовых окон Tag. Form Designer может присваивать свойству Tag исключительно строковые значения. Однако, поскольку мы определяем значение этого свойства программным путем, мы обладаем намного большими возможностями при работе с ним, и в данном случае оказывается гораздо более удобным присвоить свойству Tag логическое значение.

В конструктор добавляются следующие операторы:

```
this.btnOK.Enabled = false;

// Значения свойства Tag, необходимые для осуществления проверок на
// предмет допустимости данных
this.txtAddress.Tag = false;
this.txtAge.Tag = false;
this.txtName.Tag = false;
this.txtOccupation.Tag = false;

// Подписка на события
this.txtName.Validating += new
System.ComponentModel.CancelEventHandler(this.txtBoxEmpty_Validating);
this.txtAddress.Validating += new
     System.ComponentModel.CancelEventHandler(this.txtBoxEmpty_Validating);
```

Если вы еще не чувствуете себя достаточно уверенно, обратитесь к главе 12, где дается полное описание событий.

Обработчик события Validating представляет собой специализированную версию стандартного обработчика System.EventHandler. Причина, по которой этому событию требуется специальный обработчик, заключается в том, что, если процедура выполнения проверки допустимости завершится неудачно, необходимо иметь какой-либо способ прекратить дальнейшее выполнение приложения. Если удастся предотвратить дальнейшее выполнение приложения, то это будет в действительности означать, что не существует возможности покинуть текстовое окно до тех пор, пока введенные данные не будут корректными. Однако в данном примере такие сильнодействующие средства не используются.

События Validating и Validated используются в сочетании со свойством CausesValidation и позволяют решать очень неприятную проблему, которая возникает при использовании событий GotFocus и LostFocus для осуществления проверки допустимости в управляющих элементах. Проблема возникает,

когда события `GotFocus` и `LostFocus` генерируются постоянно, в связи с тем, что код, осуществляющий проверку допустимости, пытается переместить фокус с одного управляющего элемента на другой, что приводит к возникновению бесконечного цикла.

Добавляемый нами обработчик события имеет следующий вид:

```
privete void txtBoxEmpty_Validating(object sender,
                                    System.ComponentModel.CancelEventArgs e)
{
   // Так как отправителем является текстовое окно TextBox, то можно
   // записать посылаемый объект в текстовое окно TextBox
   TextBox tb;
   tb = (TextBox)sender;

   // Если в текстовом окне отсутствует какой-либо текст, то мы используем
   // в качестве фона Textbox красный цвет, для того чтобы указать на
   // наличие проблемы. В качестве указания на то, что в управляющем
   // элементе содержится корректная информация, используется значение тега.
   if (tb.Text.Length == 0)
   {
      tb.BackColor = Color.Red;
      tb.Tag = false;

      // В данном случае мы не хотим прерывать обработку данных,
      // но если бы мы захотели сделать это, мы бы добавили следующую строку:
      // e.Cancel = true;
   }
   else
   {
      tb.BackColor = System.Drawing.SystemColors.Window;
      tb.Tag = true;
   }

   // В заключение мы вызываем метод ValidateAll, который
   // должен задать значение кнопки OK.
   ValidateAll();
}
```

Поскольку данный метод используется для обработки событий более чем одним текстовым окном, то нельзя с уверенностью сказать, какое именно из них осуществляет вызов этой функции. Однако заведомо известно, что эффект от вызова метода должен быть одним и тем же независимо от того, кто его вызывает, поэтому можно просто приводить отправляемый параметр к типу текстового окна и работать с ним.

Если длина текста, находящегося в текстовом окне, равняется нулю, то производится установка красного цвета фона и свойству `Tag` присваивается значение `false`. В противном случае в качестве цвета фона используется стандартный цвет окна, применяемый в Windows.

Для задания стандартного цвета управляющего элемента следует всегда использовать цвета, описанные с помощью перечислимого типа `System.Drawing.SystemColors`. Если просто установить, скажем, белый цвет, то ваше приложение приобретет весьма странный вид в том случае, если пользователь изменит цвета, используемые по умолчанию.

Функция `ValidateAll()` описывается в конце настоящего раздела.

Validating — это обработчик событий, который предназначается для текстового окна Occupation. Процедура добавления аналогична двум предыдущим обработчикам событий, однако сам код, осуществляющий проверку допустимости, несколько отличается, поскольку допустимым род занятий будет считаться только в том случае, если будет введена строка Programmer или пустая строка. Следовательно, необходимо добавить в конструктор еще одну строку.

```
this.txtOccupation.Validating += new
    System.ComponentModel.CancelEventHandler(this.txtOccupation_Validating);
```

А вот как выглядит сам обработчик события:

```
private void txtOccupation_Validating(object sender,
                                      System.ComponentModel.CancelEventArgs e)
{
   // Приведение отправляемого объекта к типу Textbox
   TextBox tb = (TextBox)sender;

   // Проверка корректности значений
   if (tb.Text.CompareTo("Programmer") == 0 || tb.Text.Length == 0)
   {
      tb.Tag = true;
      tb.BackColor = System.Drawing.SystemColors.Window;
   }
   else
   {
      tb.Tag = false;
      tb.BackColor = Color.Red;
   }

   // Задание состояния кнопки OK
   ValidateAll();
}
```

Последняя проблема, которую требуется решить, связана с текстовым окном, в который вводится возраст. Налагается требование, что пользователь не должен вводить в него ничего, кроме положительных чисел, включая ноль, для упрощения проверок). Для достижения такого эффекта следует воспользоваться событием KeyPress, что позволит отбрасывать нежелательные символы до того, как они будут выведены в текстовом окне.

Во-первых, необходимо подписаться на событие KeyPress. Это выполняется в конструкторе точно таким же образом, как и в случае предшествующих обработчиков событий:

```
this.txtAge.KeyPress += new
        System.Windows.Forms.KeyPressEventHandler(this.txtAge_KeyPress);
```

Этот обработчик событий также является узкоспециализированным. В данном случае System.Windows.Forms.KeyPressEventHandler используется в качестве обработчика событий, поскольку для обработки события требуется информация о том, какая именно клавиша нажата.

А вот как выглядит собственно обработчик события:

```
private void txtAge_KeyPress(object sender,
                             System.Windows.Forms.KeyPressEventArgs e)
{
if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8)
   e.Handled = true;                                    // Удаление символа
}
```

Значения символов от 0 до 9 в кодировке ASCII лежат в диапазоне между 48 и 57, поэтому необходимо убедиться в том, что введенный символ также находится в этом интервале. Существует, однако, исключение. Так как клавише *Backspace* в кодировке ASCII соответствует значение 8, для целей редактирования его можно опустить.

Присваивание значения `true` свойству `Handled` аргумента `KeyPressEventArgs` означает, что управляющий элемент не должен больше выполнять никаких действий с данным символом, поэтому он и не будет выводиться.

На настоящий момент управляющий элемент никак не помечен не предмет допустимости или недопустимости. Это сделано по той причине, что необходимо проверить, вводилось ли вообще что-либо. Эта задача не представляет никакой сложности, поскольку ранее уже написан метод, осуществляющий подобную проверку, и требуется просто подписаться еще и на событие `Validating` для управляющего элемента `Age`, добавив следующую строчку к конструктору:

```
this.txtAge.Validating += new
      System.ComponentModel.CancelEventHandler(this.txtBoxEmpty_Validating);
```

Напоследок необходимо принять во внимание еще один случай, который касается сразу всех управляющих элемеитов. Если пользователем был введен допустимый текст во все окна, а затем внесены какие-нибудь изменения, которые вновь сделали текст недопустимым, то кнопка OK все равно останется в рабочем состоянии. В этом случае потребуется еще один обработчик событий для всех текстовых окон — обработчик события `Change` (изменение), отключающий кнопку OK, если любое текстовое поле содержит недопустимые данные.

Событие `Change` генерируется при внесении любых изменений в текст управляющего элемента. Для того чтобы подписаться на это событие, необходимо добавить следующие строки в конструктор:

```
this.txtName.TextChanged += new System.EventHandler(this.txtBox_TextChanged);
this.txtAddress.TextChanged += new
                                System.EventHandler(this.txtBox_TextChanged);
this.txtAge.TextChanged += new System.EventHandler(this.txtBox_TextChanged);
this.txtOccupation.TextChanged += new
                                System.EventHandler(this.txtBox_TextChanged);
```

Данное событие `Change` использует стандартный обработчик событий, хорошо известный нам по событию `Click`. В завершение добавляется собственно событие:

```
private void txtBox_TextChanged(object sender, System.EventArgs e)
{
   // Приведение отправляемого объекта к типу Textbox
   TextBox tb = (TextBox)sender;

   // Осуществление проверки на предмет допустимости данных
   // и установка соответствующего цвета фона.
   if (tb.Text.Length == 0 && tb != txtOccupation)
   {
      tb.Tag = false;
      tb.BackColor = Color.Red;
   }
   else if (tb == txtOccupation &&
            (tb.Text.Length != 0 && tb.Text.CompareTo("Programmer") != 0))
   {
      // Здесь не следует задавать цвет, поскольку это приведет к
      // изменению цвета в тот момент, когда пользователь будет набирать текст.
      tb.Tag = false;
   }
```

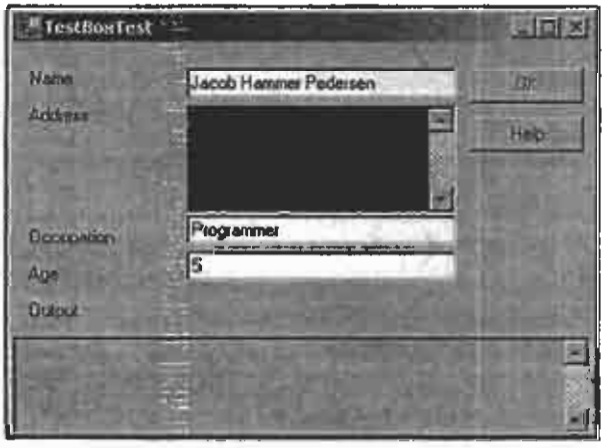
```
        else
        {
            tb.Tag = true;
            tb.BackColor = SystemColors.Window;
        }

        // Вызов метода ValidateAll для задания состояния кнопки OK
        ValidateAll();
    }
```

На этот раз требуется точно определить, какой именно управляющий элемент обращается к обработчику событий, поскольку не нужно, чтобы цвет фона окна `Occupation` изменялся на красный в тот момент, когда пользователь начинает осуществлять ввод. Для этого выполняется проверка свойства `Name` текстового окна, которое было передано в виде параметра.

Теперь осталось объяснить метод `ValidateAll()`, который делает кнопку OK работоспособной или неработоспособной:

```
    private void ValidateAll()
    {
        // Кнопка OK переводится в рабочее состояние в том случае, если
        // значение всех тегов равно true
        this.btnOK.Enabled = ((bool)(this.txtAddress.Tag) &&
                              (bool)(this.txtAge.Tag) &&
                              (bool)(this.txtName.Tag) &&
                              (bool)(this.txtOccupation.Tag));
    }
```



Этот метод просто присваивает значение `true` свойству `Enabled` в том случае, если значение каждого свойства `Tag` также имеет значение `true`. Нам необходимо изменить тип значений свойств `Tag` на логический, поскольку они хранятся в виде объекта.

При попытке протестировать программу вы увидите что-то, подобное показанному на рисунке слева.

Обратите внимание на то, что существует возможность нажимать на кнопку Help в тот момент, когда вы находитесь в текстовом окне с недопустимыми данными, и при этом цвет фона этого окна не становится красным.

Пример, рассмотрение которого только что завершено, является более длинным, чем остальные примеры, которые встретятся в настоящей главе. Причина этого чрезвычайно проста: теперь данный пример будет использоваться в качестве основы.

Не забудьте, что вы имеете возможность переписать код примеров, приведенных в настоящей книге, по адресу www.wrox.com.

Управляющие элементы RadioButton и CheckBox

Управляющие элементы `RadioButton` (переключатель) и `CheckBox` (окно для выбора) имеют общее с управляющим элементом `Button` происхождение, хотя их внешний вид и использование существенно от этого элемента отличаются.

Переключатели традиционно представлены в виде надписи с точкой, расположенной от них с левой стороны, которые могут быть либо выбраны, либо не выбраны. Переключатели рекомендуется использовать, когда необходимо предоставить пользователю возможность выбора из нескольких взаимоисключающих вариантов. Пример — запрос пола пользователя.

Для того чтобы сгруппировать переключатели в единую логическую единицу, следует использовать управляющий элемент `GroupBox`. Расположив в форме сначала групповое окно, а затем необходимые переключатели внутри границ этого группового окна, можно добиться того, что все переключатели будут изменять свое состояние таким образом, что только один из них может быть выбрана в рамках данной группы. Если не разместить их внутри группового окна, то это приведет к тому, что только один из них может быть выбрана в каждый конкретный момент времени во всей форме.

Управляющий элемент `CheckBox` традиционно представляет собой надпись с маленьким окошком с флажком, которое располагается слева от нее. Этот управляющий элемент следует использовать, когда необходимо предоставить пользователю возможность выбора одной или нескольких возможностей. Пример — вопрос, задаваемый пользователю, с какими операционными системами он знаком (например, Windows 95, Windows 98, Linux, Max OS X и т. д.).

Ниже будут рассмотрены наиболее важные свойства и события названных двух управляющих элементов, начав со управляющего элемента `RadioButton`, а затем приведены небольшие примеры, демонстрирующие их использование.

Свойства управляющего элемента RadioButton

Поскольку данный управляющий элемент является производным от класса `ButtonBase`, и мы уже сталкивались с ним в одном из предыдущих примеров, в котором использовалась кнопка, нам остается описать всего лишь несколько свойств. Как обычно, в случае необходимости ознакомиться с полным списком, обращайтесь к библиотеке MSDN:

Имя	Режим доступа	Описание
`Appearance`	Чтение/запись	Управляющий элемент `RadioButton` может выводиться в виде надписи с круглым флажком выбора, расположенным слева, в середине или справа от нее, либо в виде стандартной кнопки. В последнем случае кнопка выглядит утопленной, если она выбрана, и выпуклой — если нет.
`AutoCheck`	Чтение/запись	Когда данному свойству присвоено значение `true`, то в момент щелчка мышью на переключателе выводится флажок выбора. Если значение этого свойства равно `false`, то флажок выбора по умолчанию не выводится.
`CheckAlign`	Чтение/запись	Позволяет изменять местоположение переключателя. Он может располагаться справа, по центру или слева.
`Checked`	Чтение/запись	Определяет состояние управляющего элемента. Его значение равно `true`, если у данного управляющего элемента установлен флажок выбора, и `false` в противном случае.

События управляющего элемента RadioButton

Обычно при работе со управляющими элементами RadioButton приходится использовать только одно событие, хотя существует также большое количество других событий, на которые можно подписаться. В данной главе будут рассматриваться только два из них, причем причина, по которой второе событие упоминается,— это наличие между ними некоторого весьма тонкого отличия, на которое необходимо обратить внимание:

Имя	Событие
CheckChanged	Сообщение о наступлении этого события отправляется в тот момент, когда флажок выбора RadioButton изменяет свое состояние. Если в рамках формы или группового окна существует более одного управляющего элемента RadioButton, то сообщение об этом событии будет отправляться дважды: сначала управляющему элементу, который до этого был выбран и теперь таковым не является, а затем управляющему элементу, который выбран в настоящий момент.
Click	Сообщение о наступлении этого события отправляется всякий раз, когда происходит щелчок мышью на RadioButton. Это совсем не то же самое, что событие, связанное с изменением состояния управляющего элемента, поскольку щелчок мышью на каком-либо управляющем элементе два или более раз подряд приводит только к однократному изменению свойства "выбрано" — и то только в том случае, если оно не было выбрано до этого.

Свойства управляющего элемента CheckBox

Свойства и события данного управляющего элемента очень напоминают свойства и события RadioButton, однако они обладают двумя новыми свойствами:

Имя	Режим доступа	Описание
CheckState	Чтение/запись	В отличие от RadioButton, CheckBox может находиться в одном из трех состояний: • Checked (выбрано) • Indeterminate (не определено) • Unchecked (не выбрано) Если состояние окна выбора описывается как Indeterminate, то окно выбора, соответствующее данной надписи, обычно изображается серым цветом, что означает, что текущее значение в окне выбора либо недопустимо, либо лишено смысла при данных обстоятельствах. Пример такого состояния — ситуация, когда в Windows Explorer выбираются несколько файлов для просмотра их свойств. Если некоторые файлы открыты только для чтения, а некоторые — нет, то окна выбора, доступные только в режиме чтения, будут помечены, но изображены серым цветом — их состояние не определено.
ThreeState	Чтение/запись	Когда это свойство имеет значение false, то нет возможности изменять состояние CheckBox на Indeterminate. Хотя при этом по-прежнему можно устанавливать данное значение программным путем.

События управляющего элемента CheckBox

Обычно для данного управляющего элемента используются одно или два события. Обратите внимание на то, что, хотя событие `CheckChanged` существует и для управляющего элемента `RadioButton`, и для управляющего элемента `CheckBox`, эффекты от названных событий различны:

Имя	Описание
`CheckChanged`	Наступает при каждом изменении значения свойства `Checked` окна с выбором. Обратите внимание на то, что для управляющего элемента `CheckBox`, у которого свойство `ThreeState` имеет значение `true`, существует возможность щелкать мышью на окне выбора без изменения значения свойства `Checked`. Это происходит, когда окно выбора изменяет свое состояние с "выбрано" на "не определено".
`CheckStateChanged`	Наступает при каждом изменении значения свойства `CheckedState`. Поскольку и значение `Checked`, и значение `Unchecked` являются допустимыми значениями свойства `CheckedState`, то это сообщение об этом событии будет отправляться каждый раз, когда значение свойства `Checked` изменяется. Кроме того, оно также будет отправляться и в том случае, если состояние изменяется со значения `Checked` на значение `Indeterminate`.

Вот и все, что касается событий и свойств управляющих элементов `RadioButton` и `CheckBox`. Однако, прежде чем перейти к рассмотрению примера их использования, познакомимся с упоминавшимся ранее управляющим элементом `GroupBox`.

Управляющий элемент GroupBox

Элемент, управляющий группой окон часто используется совместно со управляющими элементами `RadioButton` и `CheckBox` для вывода вокруг них рамки с заголовком вверху — с теми из них, которые связаны между собой какими-либо логическими отношениями.

Использование элемента, управляющего группой окон не представляет никакой сложности — достаточно поместить его в форму, а затем перенести в него те управляющие элементы, которые должны в нем содержаться (но не в обратном порядке — возможность объединить в группу какие-либо заранее существующие управляющие элементы отсутствует). Результатом этого является то, что управляющие элементы оказываются потомками группового окна, а не всей формы в целом, поэтому возникает возможность иметь в каждый конкретный момент времени более одного выбранного управляющего элемента `RadioButton`. Однако внутри группового окна по-прежнему можно выбирать только один управляющий элемент `RadioButton`.

Взаимоотношения между родителями и потомками нуждаются в более подробном объяснении. Когда некоторый управляющий элемент помещается в форму, то эта форма становится для данного элемента родительской и, следовательно, сам управляющий элемент является потомком данной формы. Поскольку элемент, управляющий группой окон, сам может содержать управляющие элементы, то он считается для них родительским. Например, если переместить `GroupBox`, то переместятся все управляющие элементы, расположенные в нем.

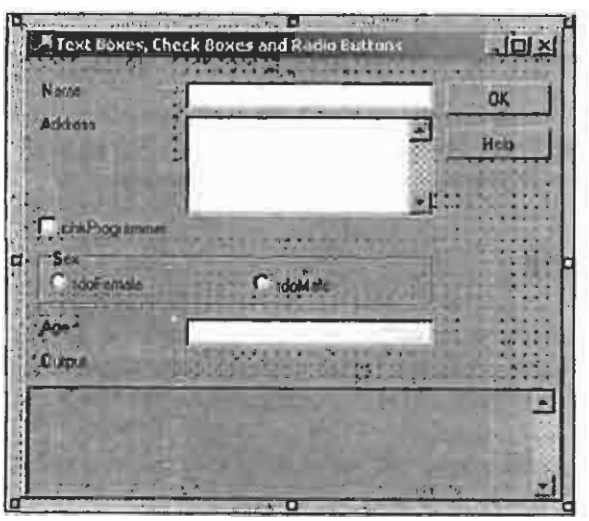
Другой результат размещения управляющих элементов внутри элемента, управляющего группой — возможность изменять значения конкретных свойств всех содержащихся в нем управляющих элементов простым изменением значения соответствующего свойства группы окон. Например, для отключения возможности использования всех управляющих элементов, находящихся в данном групповом окне, достаточно просто присвоить значение `false` свойству `Enabled` группового окна.

В приведенном ниже примере продемонстрировано использование управляющего элемента `GroupBox`.

Практикум: Пример использования управляющих элементов RadioButton и CheckBox

Мы собираемся модифицировать предыдущий пример, использовавшийся для демонстрации текстовых окон. В том примере в качестве единственного возможного рода деятельности использовалась профессия Programmer. Теперь вместо текстового окна, где нужно набрать слово Programmer, мы воспользуемся текстовым окном с выбором.

Для демонстрации возможностей `RadioButton` мы также предложим пользователю сообщить о себе дополнительную информацию — свой пол.



Внесите следующие изменения в пример использования текстовых окон:

1. Удалите надпись с именем `lblOccupation` и текстовое окно `txtOccupation`.
2. Измените размер формы, для того чтобы в ней могло разместиться групповое окно с информацией о поле пользователя, и поименуйте новые управляющие элемента так, как показано на рисунке слева.
3. Текст в управляющих элементах `RadioButton` и `CheckBox` должен совпадать с именами управляющих элементов за исключением первых трех букв.
4. Присвойте свойству `Checked` окна с выбором `chkProgrammer` значение `true`.
5. Присвойте свойству `Checked` одного из управляющих элементов — `rdoMale` или `rdoFemale` — значение `true`. Обратите внимание на то, что невозможно присвоить значение `true` сразу обоим элементам. Если вы попытаетесь сделать это, то значение второго управляющего элемента `RadioButton` автоматически изменится на `false`.

   Для видимой части примера больше делать ничего не требуется, однако необходимо еще внести определенное количество изменений в код. Во-первых, следует убрать из кода все обращения к удаленному текстовому окну. Перейдите к коду и выполните следующие действия.
6. В конструкторе формы удалите две строки, которые ссылаются на `txtOccupation`. В одной из них осуществляется подписка на события `Validating` и `TextChanged`, а другой свойству `Tag` управляющего элемента `txtBox` присваивается значение `false`.
7. Удалите метод `txtOccupation_Validating` целиком.

Этот пример находится среди программ, предназначенных для данной главы: это проект Visual Studio.NET под названием `RadioButtonAndCheckBox`.

Добавление обработчиков событий

Метод `txtBox_TextChanged` включает в себя проверки на предмет того, что вызывающим управляющим элементом было текстовое окно `txtOccupation`. Теперь нам точно известно, что это невозможно, поэтому мы внесем в этот метод некоторые изменения — удалим блок `else if` и изменим проверяемое условие `if` следующим образом:

```
private void txtBox_TextChanged(object sender, System.EventArgs e)
{
   // Приведение отправляемого объекта к типу Textbox
   TextBox tb = (TextBox)sender;

   // Осуществление проверки на предмет допустимости данных
   // и установка соответствующего цвета фона.
   if (tb.Text.Length == 0)
   {
      tb.Tag = false;
      tb.BackColor = Color.Red;
   }
   else
   {
      tb.Tag = true;
      tb.BackColor = SystemColors.Window;
   }

   // Вызов метода ValidateAll для задания состояния кнопки OK
   ValidateAll();
}
```

Последняя проверка содержимого удаленного нами текстового окна проводится с помощью метода `ValidateAll()`. Необходимо удалить всю эту проверку целиком, после чего код принимает следующий вид:

```
private void ValidateAll()
{
   // Кнопка OK переводится в рабочее состояние в том случае, если
   // значение всех тегов равно true.
   this.btnOK.Enabled = ((bool)(this.txtAddress.Tag) &&
                         (bool)(this.txtAge.Tag) &&
                         (bool)(this.txtName.Tag));
}
```

Поскольку вместо обычного текстового окна здесь используется текстовое окно с проверкой, мы можем быть уверены, что пользователь не сможет ввести никакой недопустимой информации — он либо будет программистом, либо нет.

Также известно, что пол пользователя является либо мужским, либо женским, и, поскольку свойству одного из управляющих элементов `RadioButton` присвоено значение `true`, тем самым предотвращена возможность выбора пользователем некорректного значения. Таким образом, все, что теперь необходимо сделать,— это изменить текст помощи и вывод. Это делается в обработчиках событий соответствующих кнопок:

```
private void btnHelp_Click(object sender, System.EventArgs e)
{
   // Вывод краткого описания каждого текстового окна в Output TexBox
   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.txtOutput.Text = output;
    }
```

Изменяется только текст помощи, поэтому в этом методе нет никаких неожиданностей. Ситуация с методом OK оказывается несколько более интересной:

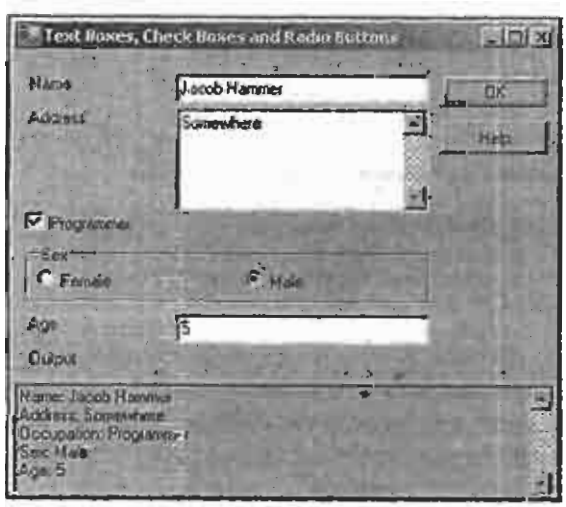
```
    private void btnOK_Click(object sender, System.EventArgs e)
    {
        // Никаких проверок на предмет недопустимости введенных значений
        // не производится, поскольку в этом нет необходимости

        string output;

        // Конкатенация текстовых значений из всех четырех текстовых окон
        output = "Name: " + this.txtName.Text + "\r\n";
        output += "Address: " + this.txtAddress.Text + "\r\n";
        output += "Occupation: " + (string)(this.chkProgrammer.Checked ?
                                    "Programmer" : "Not a programmer") + "\r\n";
        output += "Sex: " + (string)(this.rdoFemale.Checked ? "Female" :
                                                              "Male") + "\r\n";
        output += "Age: " + this.txtAge.Text;

        // Вставка нового текста
        this.txtOutput.Text = output;
    }
```

Первая из вновь включенных строк (выделены) представляет собой строку с указанием рода занятий пользователя. Мы проверяем свойство `Checked` текстового окна, и если оно равно `true`, то выводим строку Programmer, а если оно равно `false`, то выводится строка Not a programmer.



В следующей строке осуществляется проверка только переключателя `rdoFemale`. Если ее свойство `Checked` имеет значение, равное `true`, то пользователь — женщина. Если же названное значение равно `false`, то пользователь — мужчина. Поскольку в данном случае существует всего два возможных варианта, не требуется выполнять проверку второго переключателя, так как значение его свойства `Checked` всегда противоположно значению этого свойства первого.

При использовании более двух переключателей нам пришлось бы просматривать их в цикле до тех пор, пока мы бы не обнаружили тот из них, у которого значение свойства `Checked` равно `true`.

Если теперь запустить этот пример, то можно получить результаты, аналогичные изображенным на рисунке слева.

Управляющий элемент RichTextbox

Так же как и обычный управляющий элемент `Textbox`, управляющий элемент `RichTextbox` является производным от `TextBoxBase`. По этой причине он имеет несколько общих возможностей с `Textbox`, хотя они более разнообразны. Если

`Textbox` обычно используется просто с целью получения от пользователя коротких строк текста, то `RichTextbox` — для вывода и приема форматированного текста (например, выделенного **жирным шрифтом**, подчеркиванием и *курсивом*). Все это осуществляется с использованием стандарта, который называется Rich Text Format (текстовый формат с богатыми возможностями), или RTF.

В предыдущем примере мы использовали стандартный `Textbox`. С тем же успехом мы могли воспользоваться для этих целей и `RichTextbox`. Фактически, существует возможность — как можно будет увидеть из следующего примера — удалить `Textbox` с именем `txtOutput` и вставить на его место `RichTextbox` с тем же самым именем, и работа программы не претерпит при этом никаких изменений.

Свойства управляющего элемента RichTextbox

Можно предположить, что раз эта разновидность текстовых окон обладает большими возможностями, чем исследовавшаяся ранее, то она должна обладать новыми свойствами, которыми мы сможем воспользоваться, и это правда. Ниже приводится перечень наиболее часто используемых свойств `RichTextbox` с их описанием:

Имя	Режим доступа	Описание
`CanRedo`	Только чтение	Имеет значение `true` в том случае, если какое-либо действие было отменено, но может быть выполнено вновь, и `false` — в противном случае.
`CanUndo`	Только чтение	Имеет значение `true` в том случае, если существует возможность отменить действие, выполненное в `RichTextbox`, и `false` — в противном случае.
`RedoActionName`	Только чтение	Содержит в себе название действия, которое должно использоваться для повторного выполнения отмененного в `RichTextbox` действия.
`DetectUrls`	Чтение/запись	Этому свойству следует присвоить значение `true` в том случае, если требуется, чтобы управляющий элемент имел возможность определять в тексте адреса URL и форматировать их (как и в браузере — с помощью подчеркивания).
`Rtf`	Чтение/запись	Соответствует свойству `Text` за исключением того, что текст, хранящийся в нем, представлен в формате RTF.
`SelectedRtf`	Чтение/запись	Это свойство следует использовать для определения выбранного текста или выбора текста, представленного в данном управляющем элементе в формате RTF. Если такой текст будет скопирован в другое приложение, например, MS Word, то все особенности его формата будут сохранены.
`SelectedText`	Чтение/запись	Подобно свойству `SelectedRtf`, это свойство может использоваться для того чтобы определять выбранный текст или выбирать текст в данном управляющем элементе. Однако, в отличие от варианта свойства с использованием RTF, все особенности форматирования будут утрачены.
`SelectionAlignment`	Чтение/запись	Определяет способ выравнивания выбранного текста. Оно может принимать следующие значения: `Center` (по центру), `Left` (по левому краю) и `Right` (по правому краю).

Продолжение таблицы

Имя	Режим доступа	Описание
SellectionBullet	Чтение/запись	Может использоваться либо для определения того факта, что выбранный текст отформатирован с использованием буллетов, либо для вставки и удаления самих буллетов.
BulletIndent	Чтение/запись	Используется для задания числа пикселей, на которые буллеты должны быть отодвинуты.
SelectionColor	Чтение/запись	Позволяет изменять цвет выбранного текста.
SelectionFont	Чтение/запись	Позволяет изменять шрифт выбранного текста.
SelectionLength	Чтение/запись	Позволяет либо задавать, либо выбирать длину текста.
SelectionType	Только чтение	Хранит информация о том, что выбрано. Позволяет определить, что выбран один или более объектов OLE или что выбран только текст.
ShowSelectionMargin	Чтение/запись	Если этому свойству присвоить значение `true`, то слева от управляющего элемента `RichTextbox` будут выводиться поля. Это облегчит пользователю процедуру выбора текста.
UndoActionName	Только чтение	Позволяет получить имя действия, которое будет выполняться в том случае, если пользователь решит отменить какое-либо действие.
SelectionProtected	Чтение/запись	Позволяет указать, что определенные части текста не допускают внесения изменений, для чего свойству необходимо присвоить значение `true`.

Как можно заметить из приведенного перечня, большинство новых свойств имею дело с выбранным текстом. Это объясняется тем, что любое форматирование текста, которое приходится выполнять пользователю при работе со своим текстом, должно выполняться, скорее всего, только для выбранного самим пользователем участка данного текста. Если выбранный текст отсутствует, процедура форматирования начинается с той точки текста, в которой располагается курсор, известной под названием точки вставки.

События управляющего элемента RichTextbox

Большинство событий, используемых в `RichTextbox`, те же самые, что используются и в `Textbox`. Существует, однако, несколько новых событий, представляющих определенный интерес:

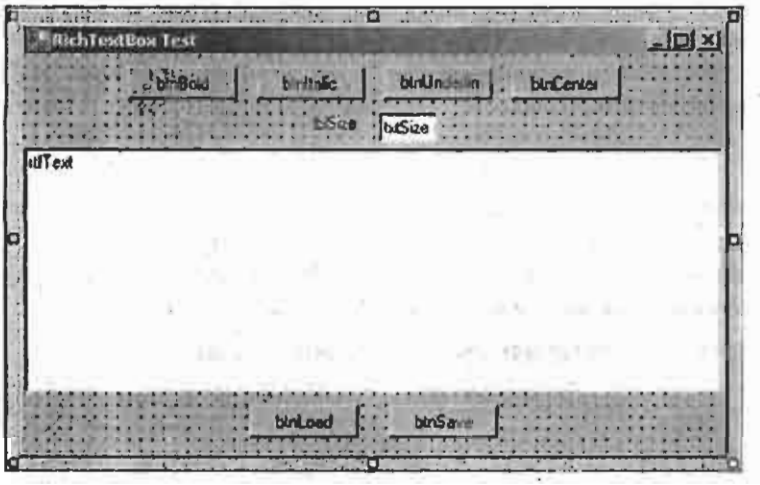
Имя	Описание
LinkedClick	Сообщение об этом событии отправляется в том случае, если пользователь щелкает мышью на ссылке, находящейся внутри текста.
Protected	Сообщение об этом событии отправляется, если пользователь пытается изменить текст, которые помечен, как защищенный.
SelectionChanged	Сообщение об этом событии отправляется, если происходит изменение выбранного текста. Если по какой-либо причине необходимо запретить пользователю менять выбранный текст, то это изменение может быть предотвращено именно здесь.

Практикум: пример использования управляющего элемента RichTextbox

В данном примере мы создадим очень примитивный текстовый редактор. Он позволит продемонстрировать, каким образом можно изменять основной формат текста и как можно сохранять и загружать текст из `RichTextbox`. Для упрощения примера текст будет загружаться и сохраняться в файле с фиксированным именем.

Начнем с разработки формы:

1. Создайте новый проект C# Windows Application и назовите его `RichTextboxTest`.
2. Создайте форму, изображенную на следующем рисунке. Текстовое окно с именем `txtSize` должно представлять собой управляющий элемент `Textbox`, а текстовое окно с именем `rtfText` должно представлять собой управляющий элемент `RichTextbox`.



3. Присвойте имена управляющим элементам, как указано на рисунке слева, и обнулите свойство `Text` как `rtfText`, так и `txtSize`.
4. Присвойте свойству `Text` всех управляющих элементов (за исключением текстовых окон) текст, который совпадает с их именами за исключением первых трех символов.
5. Присвойте свойству `Text` управляющего элемента `txtSize` значение 10.
6. Закрепите местоположение управляющих элементов в соответствии с таблицей справа.

Имя управляющего элемента	Способ закрепления
`btnLoad` и `btnSave`	`Bottom`
`rtfText`	`Top, Left, Bottom, Right`
Все остальные	`Top`

7. Задайте такое же значение свойства формы `MinimumSize`, как и у свойства `Size`.

Добавление обработчиков событий

На этом завершается работа с видимой частью приложения, и мы переходим непосредственно к коду. Для того чтобы добавить в код обработчик события `Click`, щелкните мышью два раза на кнопке Bold. Вот какой код используется для обработки данного события:

```
private void btnBold_Click(object sender, System.EventArgs e)
{
   Font oldFont;
   Font newFont;

   // Определение шрифта, который используется в выбранном тексте
   oldFont = this.rtfText.SelectionFont;
```

```
    // Если в настоящий момент текст выводится жирным шрифтом,
    // то от этого следует отказаться
    if (oldFont.Bold)
       newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
    else
       newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

    // Включение нового шрифта и активизация RichTextBox
    this.rtfText.SelectionFont = newFont;
    this.rtfText.Focus();
}
```

Работа кода начинается с определения используемого в выбранном в настоящий момент тексте шрифта и присвоении локальной переменной `oldFont`. Затем производится проверка, не является ли текущий текст уже выделенным жирным шрифтом. Если является, то необходимо убрать выделение текста жирным шрифтом; в противном случае требуется выделить текст жирным шрифтом. Мы создаем новый шрифт, используя переменную `oldFont` в качестве прототипа и добавляя и удаляя выделение жирным шрифтом в зависимости от необходимости.

В заключение мы присванваем выбранному тексту новый шрифт и возвращаем фокус управляющему элементу `RichTextbox`.

Объект Font (шрифт) описывается в главе 16.

Обработчики событий для управляющих элементов `btnItalic` и `btnUnderline` совершенно аналогичны приведенному выше, за исключением того, что в них выполняется проверка наличия соответствующего стиля. Дважды щелкните мышью на кнопках Italic и Underline и добавьте следующий код:

```
private void btnItalic_Click(object sender, System.EventArgs e)
{
    Font oldFont;
    Font newFont;

    // Определение шрифта, который используется в выбранном тексте
    oldFont = this.rtfText.SelectionFont;

    // Если в настоящий момент текст выделяется курсивом,
    // то от этого следует отказаться
    if (oldFont.Italic)
       newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
    else
       newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

    // Включение нового шрифта
    this.rtfText.SelectionFont = newFont;
    this.rtfText.Focus();
}

private void btnUnderline_Click(object sender, System.EventArgs e)
{
    Font oldFont;
    Font newFont;

    // Определение шрифта, который используется в выбранном тексте
    oldFont = this.rtfText.SelectionFont;

    // Если в настоящий момент текст выделяется подчеркиванием,
    // то от этого следует отказаться
    if (oldFont.Underline)
       newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
    else
       newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);
```

```
        // Включение нового шрифта
        this.rtfText.SelectionFont = newFont;
        this.rtfText.Focus();
    }
```

Теперь дважды щелкните мышью на последней из ответственных за форматирование текста кнопке — Center (выравнивание по центру) — и добавьте следующий код:

```
    private void btnCenter_Click(object sender, System.EventArgs e)
    {
        if (this.rtfText.SelectionAlignment == HorizontalAlignment.Center)
            this.rtfText.SelectionAlignment = HorizontalAlignment.Left;
        else
            this.rtfText.SelectionAlignment = HorizontalAlignment.Center;
        this.rtfText.Focus();
    }
```

Теперь необходимо проверить свойство `SelectionAlignment` на предмет того, не является ли выбранный текст уже выровненным по центру. `HorizontalAlignment` (выравнивание по горизонтали) — это перечислимый тип, который может принимать следующие значения: `Left`, `Right`, `Center`, `Justify` (по формату) и `Notset` (не задано). В данном случае выполняется проверка на предмет того, на равно ли значение `Center`, и если равно, то выполняется выравнивание по левому краю. Если нет — задается выравнивание по центру.

Последний вид форматирования, на который способен наш небольшой текстовый редактор,— это установка размера текста. Для текстового окна `Size` мы добавим два обработчика событий: для управления вводом и для определения момента, когда пользователь завершает ввод значения.

Добавьте следующие строки в конструктор формы:

```
    public Form1()
    {
        InitializeComponent();

        // Подписка на события
        this.txtSize.KeyPress += new
            System.Windows.Forms.KeyPressEventHandler(this.txtSize_KeyPress);
        this.txtSize.Validating += new
            System.ComponentModel.CancelEventHandler(this.txtSize_Validating);
    }
```

Эти два обработчика событий уже встречались в предшествующем примере. Оба события используют вспомогательный метод, называемый `ApplyTextSize`, которому передается строка текста с указанием его размера:

```
private void txtSize_KeyPress(object sender,
                              System.Windows.Forms.KeyPressEventArgs e)
{
    // Удаление всех символов, которые не являются цифрами, пробелом или вводом
    if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8 && e.KeyChar != 13)
    {
        e.Handled = true;
    }
    else if (e.KeyChar == 13)
    {
        // Использование заданного размера в том случае, если пользователь
        // нажимает клавишу Enter
        TextBox txt = (TextBox)sender;
```

```
        if (txt.Text.Length > 0)
            ApplyTextSize(txt.Text);
        e.Handled = true;
        this.rtfText.Focus();
    }
}

private void txtSize_Validating(object sender,
                                System.ComponentModel.CancelEventArgs e)
{
    TextBox txt = (TextBox)sender;

    ApplyTextSize(txt.Text);
    this.rtfText.Focus();
}

private void ApplyTextSize(string textSize)
{
    // Преобразование размера текста в тип с плавающей точкой, поскольку
    // вскоре нам потребуется именно этот тип
    float newSize = Convert.ToSingle(textSize);
    FontFamily currentFontFamily;
    Font newFont;

    // Создание шрифта того же самого семейства, но обладающего новым размером
    currentFontFamily = this.rtfText.SelectionFont.FontFamily;
    newFont = new Font(currentFontFamily, newSize);

    // Присваивание нового значения шрифту выбранного текста
    this.rtfText.SelectionFont = newFont;
}
```

Действия, которые могут представлять для нас интерес, происходят во вспомогательном методе `ApplyTextSize`. Сначала этот метод преобразовывает размер из строкового типа в тип с плавающей точкой (float). Пользователю не позволяется вводить ничего, кроме целых значений, однако при создании нового шрифта требуется значение типа `float`, поэтому такое преобразование необходимо.

После этого мы получаем семейство, к которому принадлежит данный шрифт, и создаем на базе этого семейства новый шрифт заданного размера. Наконец, мы присваиваем новый шрифт шрифту выбранного текста.

Это все виды форматирования, которые можно выполнять, однако некоторые возможности предоставляются самим управляющим элементом `RichTextBox`. Если вы теперь попытаетесь запустить этот пример, то вы получите возможность выделять текст жирным шрифтом, курсивом и подчеркиванием, кроме того, вы сможете выровнять текст по центру. Это те возможности, которые вы и ожидали получить, однако существуют еще некоторые возможности, представляющие интерес. Попробуйте набрать в тексте какой-нибудь web-адрес, например, `www.wrox.com`. Этот текст будет распознан управляющим элементом как интернетовский адрес и будет выделен подчеркиванием, а курсор мыши, попадая на него, будет превращаться в кисть руки. Если вы думаете, что можете теперь нажать кнопку мыши и попасть на эту страницу, то вы почти правы. Но для этого необходимо обработать событие, которое возникает, когда пользователь щелкает мышью на ссылке,— `LinkClicked`.

Это достигается посредством подписки на событие в конструкторе:

```
this.rtfText.LinkClicked += new
    System.Windows.Forms.LinkClickedEventHandler(this.rtfText_LinkedClick);
```

С этим обработчиком событий мы пока еще не сталкивались. Он используется для получения текстового представления ссылки, на которой щелкнули мышью. Этот обработчик очень прост и выглядит следующим образом:

```
private void rtfText_LinkedClick(object sender,
                                 System.Windows.Forms.LinkClickedEventArgs e)
{
   System.Diagnostics.Process.Start(e.LinkText);
}
```

Этот код позволяет открыть браузер, использующийся по умолчанию, если он еще не открыт, и перейти на сайт, на который указывает данная ссылка.

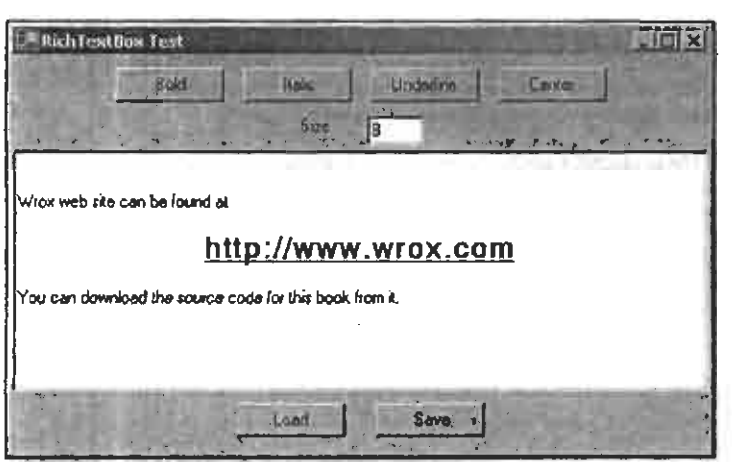
Часть приложения, отвечающая за редактирование, завершена. Все, что осталось сделать, — это загрузить и сохранить содержимое управляющего элемента. Для этой цели мы будем использовать файл с фиксированным именем.

Щелкните два раза мышью на кнопке Load и добавьте следующий код:

```
private void btnLoad_Click(object sender, System.EventArgs e)
{
   // Загрузка файла в управляющий элемент RichTextbox
   try
   {
      rtfText.LoadFile("../../Test.rtf");
   }
   catch (System.IO.FileNotFoundException)
   {
      MessageBox.Show("No file to load yet");
   }
}
```

Вот и все! Больше ничего не требуется. Поскольку мы имеем дело с файлами, то всегда существует вероятность столкнуться с исключительными ситуациями, и мы должны уметь их обрабатывать. В методе `Load` мы обрабатываем исключительную ситуацию, которая возникает в том случае, если файла с таким именем не существует. Не менее просто и сохранить файл. Щелкните мышью два раза на кнопке **Save** и добавьте следующее:

```
private void btnSave_Click(object sender, System.EventArgs e)
{
   // Сохранение текста
   try
   {
      rtfText.SaveFile("../../Test.rtf");
   }
   catch (System.Exception err)
   {
      MessageBox.Show(err.Message);
   }
}
```



Теперь попробуйте запустить пример и нажмите кнопку **Save**. Очистите текстовое окно и нажмите кнопку **Load** — текст, который вы только что сохранили, должен появиться вновь.

На этом пример использования управляющего элемента `RichTextbox` завершен. Если вы запустите его, то окажетесь в состоянии создать нечто вроде того, что изображено на рисунке слева.

Управляющие элементы ListBox и CheckedListBox

Окна со списком используются для вывода списка строк, из которых одновременно может быть выбрана одна или несколько строк. Окно со списком точно так же, как окна с выбором и переключатели, предоставляет возможность запрашивать выбор пользователя, состоящий из одного или нескольких наименований. Окна со списками следует использовать, когда в период разработки не известно фактическое количество элементов, из которых может выбирать пользователь (в качестве примера можно взять список сотрудников). Даже если на момент разработки известны все возможные значения, то использование окна со списком предпочтительно, если количество таких значений очень велико.

Класс `ListBox` является производным от класса `ListControl`, предоставляющего базовые функциональные возможности для двух управляющих элементов с использованием списка, которые стандартно поставляются с Visual Studio.NET. Второй управляющий элемент — `ComboBox` — будет рассмотрен в настоящей главе позднее.

С Visual Studio.NET поставляется еще одна разновидность окна со списком. Это окно называется `CheckedListBox` и является производным от класса `ListBox`. Этот управляющий элемент позволяет работать с точно таким же списком, как и управляющий элемент `ListBox`, однако в нем дополнительно для каждого элемента списка предусмотрено наличие окна выбора.

Свойства управляющего элемента ListBox

В списке, который приводится ниже, все свойства присущи одновременно и классу `ListBox`, и классу `CheckedListBox`, если явно не утверждается обратное:

Имя	Режим доступа	Описание
`SelectedIndex`	Чтение/запись	Значение этого свойства представляет отсчитываемый с нуля индекс выбранного элемента в окне со списком. Если окно со списком допускает одновременный выбор нескольких элементов, то это свойство содержит значение первого выбранного элемента списка.
`ColumnWidth`	Чтение/запись	В окне со списком, которое допускает наличие нескольких столбцов элементов, это свойство определяет ширину столбцов.
`Items`	Только чтение	Содержит все элементы окна со списком. Свойства этого семейства можно использовать для добавления и удаления элементов.
`MultiColumn`	Чтение/запись	Окно со списком может иметь более одного столбца. Это свойство можно использовать для получения или задания числа столбцов в данном окне.
`SelectedIndices`	Только чтение	Представляет собой семейство, в котором содержатся индексы всех выбранных элементов в окне со списком (отсчет индексов ведется с нуля).
`SelectedItem`	Чтение/запись	Если в данном окне со списком допускается выбор только одного элемента, то это свойство будет содержать выбранный элемент, если таковой существует. Если в окне допускается выбор более, чем одного элемента, оно будет содержать первый из выбранных элементов.

Продолжение таблицы

Имя	Режим доступа	Описание
`SelectedItems`	Только чтение	Представляет собой семейство, в котором содержатся все элементы, выбранные в настоящий момент.
`SelectionMode`	Чтение/запись	Существует возможность задать одни из четырех различных режимов выбора в окне со списком: • `None`: Элементы не могут быть выбранными. • `One`: В каждый конкретный момент времени допускается выбор только одного элемента. • `MultiSimple`: Допускается выбор нескольких элемеитов. • `MultiExtended`: Допускается выбор нескольких элементов, и пользователь при осуществлении своего выбора может использовать клавиши *Ctrl*, *Shift* и *стрелки*.
`Sorted`	Чтение/запись	Присваивание этому свойству значения `true` приведет к тому, что все элементы, содержащиеся в управляющем элемеите `ListBox`, будут отсортированы в алфавитном порядке.
`Text`	Чтение/запись	Свойство `Text` встречалось во многих управляющих элементах, однако в данном случае оно работает совершенно отличным образом по сравнению с тем, что мы видели до сих пор. Если свойству `Text` управляющего элемента `ListBox` присваивается некоторое значение, то оно осуществляет поиск элемента с соответствующим текстом и помечает его как выбранный. Если значение свойства `Text` считывается, то в качестве возвращаемого значения используется первый выбраиный элемеит из списка. Это свойство не может использоваться, если значение свойства `SelectionMode` равно `None`.
`ChekcedIndices`	Только чтение	(Только для `CheckedListBox`.) Представляет собой семейство, где содержатся все индексы в `CheckedListBox`, обладающие установленным флажком или находящиеся в неопределенном состоянии.
`CheckedItems`	Только чтение	(Только для `CheckedListBox`.) Представляет собой семейство всех элементов в `CheckedListBox`, которые обладают установленным флажком или находятся в неопределенном состоянии.
`CheckOnClick`	Чтение/запись	(Только для `CheckedListBox`.) Если значение этого свойства равно `true`, то названный элемент будет изменять свое состояние каждый раз, когда пользователь будет щелкать на нем мышью.
`ThreeDCheckBoxes`	Чтение/запись	(Только для `CheckedListBox`.) Значение этого свойства позволяет использовать окна выбора, которые либо выводятся плоскими, либо имеют обычный вид.

Методы управляющего элемента ListBox

Для того чтобы с окном со списком можно было эффективно работать, следует знать несколько методов, которые могут быть для этого использованы. В таблице, следующей ниже, перечисляются наиболее часто используемые методы. Если иное не утверждается явно, названные методы принадлежат и классу `ListBox`, и классу `CheckedListBox`.

Имя	Описание
ClearSelected	Помечает все выбранные элементы в ListBox как невыбранные.
FindString	Находит первое вхождение в ListBox, которое начинается с указанной вами строки. Например, FindString("a") позволит найти первую строку в ListBox, которая начинается с символа "a".
FindStringExact	Работает аналогично FindString, однако требуется совпадение полной строки.
GetSelected	Возвращает значение, которое позволяет определить, является ли данный элемент выбранным.
SetSelected	Позволяет пометить элемент как выбранный/невыбранный.
ToString	Возвращает текущий выбранный элемент.
GetItemChecked	(Только для CheckedListBox.) Возвращает значение, которое позволяет определить, является ли данный элемент выбранным или нет.
GetItemCheckState	(Только для CheckedListBox.) Возвращает значение, которое позволяет определить состояние элемента.
SetItemChecked	(Только для CheckedListBox.) Помечает указанный элемент как выбранный.
SetItemCheckState	(Только для CheckedListBox.) Устанавливает состояние элемента.

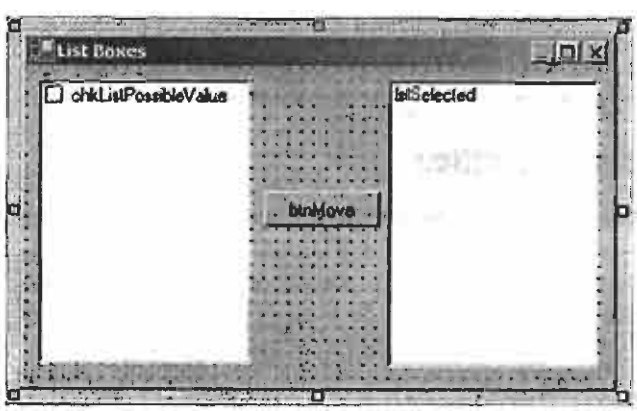
События управляющего элемента ListBox

Обычно для работы со управляющими элементами ListBox и CheckedListBox необходимо знать о наступлении событий, связанных с тем, какие элементы выбирает пользователь:

Имя	Описание
ItemCheck	(Только для CheckedListBox.) Наступает в том случае, если состояние одного из элементов списка изменяется.
SelectedIndexChanged	Наступает в тот момент, когда изменяется индекс выбранного элемента.

Практикум: пример использования управляющего элемента ListBox

Мы создадим небольшой пример, в котором будут использоваться оба управляющих элемента — ListBox и CheckedListBox. Пользователю будет предоставлена возможность выбирать элементы в CheckedListBox, а затем посредством нажатия кнопки переносить выбранные элементы в обычный ListBox. Мы создадим следующий диалог:



1. Откройте новый проект в Visual Studio.NET с именем Lists. Добавьте в форму управляющие элементы ListBox, CheckedListBox и кнопку и измените их имена, как показано на рисунке слева.
2. Замените значение свойства Text кнопки на "Move".
3. Замените значение свойства CheckOnClick управляющего элемента CheckedListBox на true.

Добавление обработчиков событий

Теперь мы готовы к написанию некоторого кода. В тот момент, когда пользователь щелкает мышью на кнопке Move, нам требуется найти выбранные элементы и скопировать их в окно со списком Selected.

Щелкните мышью два раза на кнопке Move и введите следующий код:

```
private void btnMove_Click(object sender, System.EventArgs e)
{
   // Проверка на предмет наличия каких-либо выбранных элементов в CheckedListBox
   if (this.chkListPossibleValues.CheckedItems.Count > 0)
   {
      // Очистка текстового окна, в которое будут помещаться выбранные элементы
      this.lstSelected.Items.Clear();

      // Просмотр в цикле семейства выбранных элементов CheckedItems
      // в CheckedListBox и перенос их в окно со списком Selected
      foreach (string item in this.chkListPossibleValues.CheckedItems)
      {
         this.lstSelected.Items.Add(item.ToString());
      }

      // Сброс всех выбранных элементов в CheckedListBox
      for (int i = 0; i < this.chkListPossibleValues.Items.Count; i++)
         this.chkListPossibleValues.SetItemChecked(i, false);
   }
}
```

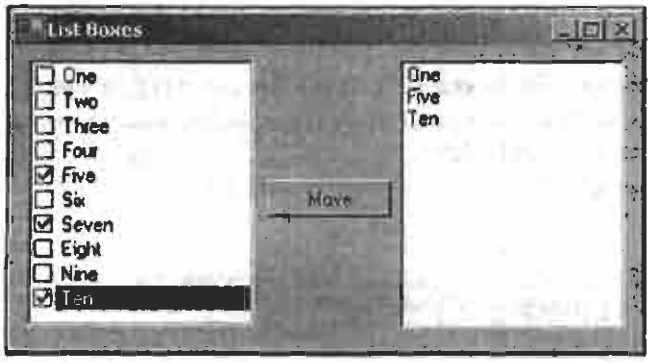
Сначала мы осуществляем проверку свойства `Count` семейства `CheckedItems`. Значение названного свойства будет больше нуля, если в данном семействе выбраны какие-либо элементы. Затем мы удаляем все элементы из окна со списком Selected и проходим в цикле по семейству `CheckedItems`, добавляя каждый элемент семейства в окно со списком Selected. В заключение мы помечаем все элементы в `CheckedListBox` как невыбранные.

Теперь необходимо занести что-либо в `CheckedListBox` — чтобы было, что переносить. Существует возможность добавлять элементы в процессе разработки, выбрав свойство `Items` на панели свойств. Мы вместо этого будем добавлять элементы программным путем (в конструкторе нашей формы):

```
public Form1()
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();

   // Заполнение CheckedListBox
   this.chkListPossibleValues.Items.Add("One");
   this.chkListPossibleValues.Items.Add("Two");
   this.chkListPossibleValues.Items.Add("Three");
   this.chkListPossibleValues.Items.Add("Four");
   this.chkListPossibleValues.Items.Add("Five");
   this.chkListPossibleValues.Items.Add("Six");
   this.chkListPossibleValues.Items.Add("Seven");
   this.chkListPossibleValues.Items.Add("Eight");
   this.chkListPossibleValues.Items.Add("Nine");
   this.chkListPossibleValues.Items.Add("Ten");
}
```

Как и для добавления элементов с помощью панели свойств, для добавления элементов в процессе выполнения используется семейство `Items`.



На этом создание примера по использованию окон со списком завершено, и если вы запустите полученный в результате ваших стараний код, то увидите что-нибудь, подобное изображенному на рисунке слева.

Управляющий элемент комбинированного окна (ComboBox)

Как и предполагает его название, комбинированное окно объединяет в себе несколько управляющих элементов, а именно `Textbox`, `Button` и `ListBox`. В отличие от `ListBox` не существует возможности ни при каких обстоятельствах выбирать более одного элемента из списка элементов, содержащихся в `ComboBox`, кроме того существует дополнительная возможность вводить новые вхождения в список в `Textbox`, являющийся составной частью `ComboBox`.

Обычно управляющий элемент `ComboBox` используется для экономии места, отводимого для диалога, поскольку та часть комбинированного окна, которая видна постоянно, состоит из текстового окна и кнопки. Когда пользователь нажимает стрелку, расположенную справа от текстового окна, то раскрывается окно со списком, в котором пользователь может осуществить свой выбор. После того как процедура выбора завершена, окно со списком сворачивается и выводимая картинка снова принимает свой обычный вид.

Мы рассмотрим свойства и события этого управляющего элемента, а затем создадим пример, в котором будут использоваться управляющие элементы `ComboBox` и два `ListBox`.

Свойства управляющего элемента ComboBox

Поскольку управляющему элементу `ComboBox` присущи черты таких элементов, как `Textbox` и `ListBox`, то многие свойства и события `ComboBox` относятся также и к этим двум управляющим элементам. По этой причине управляющий элемент `ComboBox` обладает огромным количеством свойств и событий, из которых в настоящей книге будут рассматриваться только наиболее часто используемые. Исчерпывающий перечень можно найти в библиотеке MSDN.

Имя	Режим доступа	Описание
`DropDownStyle`	Чтение/запись	При выводе комбинированного окна могут использоваться три различных стиля: • `DropDown`: Пользователь имеет возможность редактировать текстовое окно, а для того чтобы вывести на экран ту часть управляющего элемента, которая содержит список, должен щелкнуть мышью на стрелке. • `Simple`: Аналогично `DropDown` за исключением того, что та часть управляющего элемента, которая содержит список, является постоянно видимой — почти так же, как и для обычного управляющего элемента `ListBox`. • `DropDownList`: Пользователь не имеет возможности редактировать текстовое окно, а для того чтобы вывести на экран ту часть управляющего элемента, которая содержит список, должен щелкнуть мышью на стрелке.

Продолжение таблицы

Имя	Режим доступа	Описание
`DroppedDown`	Чтение/запись	Указывает на то, в каком состоянии — развернутом или свернутом — находится та часть управляющего элемента, которая содержит список. Если присвоить этому свойству значение `true`, то список будет развернут.
`Items`	Только чтение	Представляет собой семейство, в котором хранятся все элементы списка, содержащегося в комбинированном окне.
`MaxLength`	Чтение/запись	Присвоив этому свойству любое значение, не равное нулю, вы можете определить максимально допустимое символов, которые могут быть введены в часть управляющего элемента, представляющую собой текстовое окно.
`SelectedIndex`	Чтение/запись	Содержит значение индекса элемента списка, выбранного в настоящий момент.
`SelectedItem`	Чтение/запись	Содержит сам элемент, который в настоящий момент является выбранным.
`SelectedText`	Чтение/запись	Содержит текст, который выбран в текстовом окне управляющего элемента.
`SelectionStart`	Чтение/запись	Это свойство представляет собой индекс первого выбранного в текстовом окне символа.
`SelectionLength`	Чтение/запись	Определяет длину текста, выбранного в текстовом окне управляющего элемента `ComboBox`.
`Sorted`	Чтение/запись	Если этому свойству присвоить значение, равное `true`, то элементы списка будут отсортированы в алфавитном порядке.
`Text`	Чтение/запись	Если этому свойству присвоить значение, равное `null`, то любой выбор в той части управляющего элемента, которая содержит список, будет сброшен. Если присвоить ему какое-либо существующее в списке значение, то это значение становится выбранным. Если такого значения в списке не существует, то текст будет просто выведен в текстовом окне управляющего элемента `ComboBox`.

События управляющего элемента ComboBox

Существует три основных действия, происходящих в комбинированном окне, о которых вам может потребоваться уведомление:

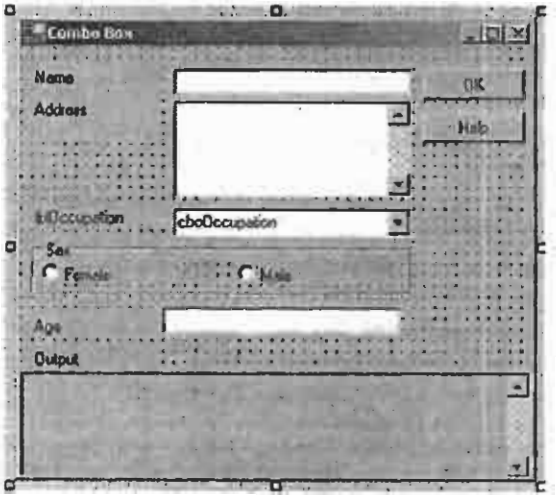
- Изменение выбора
- Изменение состояния раскрывающегося списка
- Изменения в тексте

Чтобы иметь возможность обрабатывать эти действия, обычно требуется подписаться на одно или несколько следующих событий:

Имя	Описание
`DropDown`	Возникает в момент, когда разворачивается часть управляющего элемента, содержащая список.
`SelectedIndexChanged`	Возникает в момент, когда изменяется выбор в части управляющего элемента, содержащей список.
`KeyDown` `KeyPress` `KeyUp`	Эти события происходят при нажатии клавиши в том случае, если в фокусе находится та часть управляющего элемента, в которой содержится текст. Описание этих событий можно найти выше в настоящей главе, в разделе, посвященном текстовым окнам.
`TextChanged`	Возникает в момент изменения свойства `Text`.

Практикум: пример использования управляющего элемента ComboBox

В данном случае мы снова обратимся к примеру, которым иллюстрировалось использование управляющего элемента Textbox. В примере, посвященном управляющим элементам Checkbox, мы заменили Textbox на Checkbox при запросе рода занятий. Здесь следует заметить, что существуют другие виды деятельности, не связанные с программированием, и нам потребуется изменить этот диалог еще раз, чтобы иметь возможность работать с этими видами. С этой целью мы будем использовать комбинированное окно, в котором содержатся два различных рода деятельности — консультант и программист. Кроме того, пользователю будет предоставлена возможность ввести род своей деятельности в тех случаях, если она отлична от этих двух.



Поскольку мы не хотим требовать от пользователя вводить род своей деятельности каждый раз, когда им используется данное приложение, мы будем сохранять в файле элементы списка ComboBox при каждом закрытии диалога и загружать при его открытии.

Во-первых, внесем изменения во внешний вид диалога:

1. Удалите управляющие элементы CheckBox с именем chkProgrammer.
2. Добавьте надпись и управляющий элемент ComboBox, назовите их так, как показано на рисунке слева.
3. Измените свойство Text надписи Occupation и очистите его в ComboBox.
4. Больше никаких изменений вносить в форму не требуется. Однако необходимо еще создать файл Occupations.txt, в котором содержатся следующие две строки:

```
Consultant
Programmer
```

Этот пример можно найти среди программ, относящихся к данной главе, в виде проекта Visual Studio.NET с именем ComboBox.

Добавление обработчиков событий

Теперь можно вносить изменения в код. Перед тем как приступить к созданию нового кода, необходимо модифицировать обработчик события btnOK_Click таким образом, чтобы учесть изменения, которые мы внесли в форму:

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    // Никаких проверок на предмет недопустимости введенных значений
    // не производится, поскольку в этом нет необходимости

    string output;

    // Конкатенация текстовых значений из всех четырех текстовых окон
    output = "Name: " + this.txtName.Text + "\r\n";
    output += "Address: " + this.txtAddress.Text + "\r\n";
```

```
    output += "Occupation: " + this.cboOccupation.Text + "\r\n";
    output += "Sex: " + (string)(this.rdoFemale.Checked ? "Female" : "Male") +
              "\r\n";

    output += "Age: " + this.txtAge.Text;

    // Вставка нового текста
    this.txtOutput.Text = output;
}
```

Вместо окна с выбором теперь используется комбинированное окно, поэтому теперь, когда выбирается некоторый элемент списка, он выводится в той части управляющего элемента, где есть возможность осуществлять редактирование. По этой причине интересующее нас значение всегда будет находиться в свойстве `Text` комбинированного окна.

Теперь можно создавать новый код. Первое, что необходимо сделать,— это разработать метод, который будет загружать файл с уже существующими значениями, и вносить их в комбинированное окно:

```
private void LoadOccupations()
{
   try
   {
      // Создание объекта StreamReader. Необходимо изменить путь на тот, по
      // которому можно найти соответствующий файл
      System.IO.StreamReader sr =
                  new System.IO.StreamReader("../../Occupations.txt");

      string input;

      // Чтение происходит до тех пор, пока имеются несчитанные строки
      do
      {
         input = sr.ReadLine();

         // Добавление считанной строки только в том случае, если в ней
         // содержатся какие-либо символы
         if (input != "")
            this.cboOccupation.Items.Add(input);
      } while (sr.Peek() != -1);
                                 // Peek возвращает -1 в том случае, если
                                 // достигнут конец потока

      // Закрытие потока
      sr.Close();
   }
   catch (System.Exception)
   {
      MessageBox.Show("File not found");
   }
}
```

Программа считывания из потока объясняется в главе 20, поэтому здесь мы не будем вдаваться в подробности работы этого кода. В данный момент будет достаточно сказать, что мы используем эту программу для того чтобы открыть текстовый файл `Occupations.txt` и считывать из него строки для управляющего элемента `ComboBox` по одной. Для добавления каждой считанной строки из файла в комбинированное окно нами используется метод `Add()` семейства `Items`.

Поскольку пользователю предоставлена возможность вводить новые элементы списка в комбинированное окно, добавим проверку на нажатие клавиши *Enter*.

Если текст, находящийся в свойстве Text управляющего элемента ComboBox, отсутствует в семействе Items, то мы добавляем этот новый элемент в семейство:

```
private void cboOccupation_KeyDown(object sender,
                                   System.Windows.Forms.KeyEventArgs e)
{
   int index = 0;
   ComboBox cbo = (ComboBox)sender;

   // Нам необходимо что-либо предпринимать только в том случае,
   // если была нажата клавиша Enter
   if (e.KeyCode == Keys.Enter)
   {
      // Метод FindStringExact осуществляет поиск строки и не является
      // чувствительным к регистру; что нам и требуется, т. к. Programmer
      // ничем не отличается от programmer. Если удается обнаружить
      // совпадение, то данный элемент в ComboBox становится выбранным.
      index = cbo.FindStringExact(cbo.Text);
      if (index < 0)      // Метод FindStringExact возвращает значение -1
                          // в том случае, если ничего не найдено.
         cbo.Items.Add(cbo.Text);
      else
         cbo.SelectedIndex = index;

      // Указывает на то, что событие KeyDown нами обработано.
      e.Handled = true;
   }
}
```

Метод FindStringExact() объекта ComboBox осуществляет поиск строки по полному совпадению двух строк, но без учета регистра в обеих строках. Это нас вполне устраивает, поскольку нам не требуется добавлять в семейство один и тот же вид деятельности, набранный с использованием символов в различных регистрах.

Если не удается найти в семействе Items элемент с совпадающим текстом, то добавляется новый элемент. Это автоматически приводит к тому, что он становится выбранным. Если же совпадающий элемент удается обнаружить, то мы просто выбираем уже существующее вхождение в семействе Items.

В конструкторе формы нам необходимо подписаться на события:

```
this.txtAge.TextChanged += new System.EventHandler(this.txtBox_TextChanged);
this.cboOccupation.KeyDown += new
        System.Windows.Forms.KeyEventHandler(this.cboOccupation_KeyDown);
```

В тот момент, когда пользователь закрывает диалог, необходимо сохранить элементы, находящиеся в комбинированном окне. Для этого предусмотрен еще один метод:

```
private void SaveOccupation()
{
   try
   {
      System.IO.StreamWriter sw = new
                        System.IO.StreamWriter("../../Occupations.txt");
      foreach (string item in this.cboOccupation.Items)
         sw.WriteLine(item);                    // Запишите элемент в файл
      sw.Flush();
      sw.Close();
   }
   catch (System.Exception e)
   {
      MessageBox.Show("File not found or moved");
   }
}
```

Класс `StreamWriter` будет рассматриваться в главе 20, поэтому данный код в настоящий момент подробно обсуждаться не будет. Еще раз обратите внимание на то, что код, ответственный за выполнение операции ввода/вывода файла, помещен внутрь блока `try...catch` — это мера предосторожности на тот случай, если кто-либо по невнимательности удалил или перенес текстовый файл, пока форма была открыта. Просмотр элемеитов семейства `Items` и запись каждого из них в файл осуществляется в цикле.

Наконец, иеобходимо выполнить обращение к только что определенным методам `LoadOccupation()` и `SaveOccupation()`. Это делается в конструкторе формы и в методе `Dispose()` соответственно:

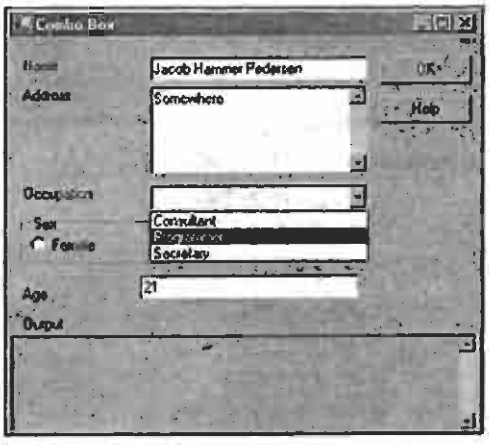
```
public Form1()
{
   .
   .
   .
   this.cboOccupation.KeyDown += new
         System.Windows.Forms.KeyEventHandler(this.cboOccupation_KeyDown);

   // Заполнение ComboBox
   LoadOccupations();
}

public override void Dispose()
{
   // Сохранение элементов в ComboBox
   SaveOccupation();

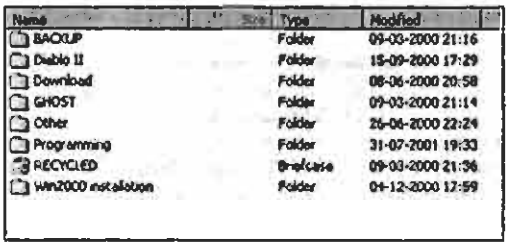
   base.Dispose();
   if (components != null)
      components.Dispose();
}
```

На этом пример использования управляющего элемента `ComboBox` завершен. Запустив этот пример, вы должны получить что-то, подобное изображенному на рисунке справа.



Управляющий элемент ListView

Список, из которого можно выбирать открываемые файлы в стандартных диалоговых окнах, используемых в Windows, представляет собой управляющий элемент `ListView` (просмотр списков). Все возможности, связанные со стандартным диалогом просмотра (большие изображения, подробный просмотр и т. д.), обеспечиваются управляющим элементом `ListView`, поставляемым вместе с Visual Studio.NET (см. рис. слева).



Просмотр списков обычно используется для представления данных, когда пользователям предоставляются определенные возможности по управлению некоторыми особенностями и стилем представления. Можно выводить содержащиеся в управляющем элементе данные в виде столбцов и строк, очень напоминающем матрицу,

в виде единственного столбца и с различным представлением изображений. Наиболее часто используемым способом представления списка является как бы вид сверху, который используется для поиска по папкам, имеющимся в компьютере.

Управляющий элемент `ListView` наверняка является наиболее сложным среди тех, с которыми нам придется столкнуться в настоящей главе, и полное рассмотрение этого управляющего элемента выходит за рамки настоящей книги. Все, что мы собираемся сделать в данной главе,— это предоставить вам солидное основание для дальнейшей работы, продемонстрировав пример, в котором используется большинство из наиболее значимых возможностей управляющего элемента `ListView`, а также подробно описав многочисленные свойства, события и методы, которые могут быть использованы. Мы также уделим некоторое внимание управляющему элементу `ImageList`, который предназначается для хранения изображений, используемых управляющим элементом `ListView`.

Свойства управляющего элемента ListView

Имя	Режим доступа	Описание
`Activation`	Чтение/запись	Используя данное свойство, можно управлять активизацией элемента при просмотре списка пользователем. Изменять установку, принятую по умолчанию, не рекомендуется до тех пор, пока у вас не появятся для этого достаточные основания, поскольку в этом случае будет изменена установка, которая была задана пользователем для всей его системы. Это свойство может принимать следующие значения: • `Standard`: Означает установку, выбранную пользователем для всего его компьютера. • `OneClick`: Элемент активизируется однократным щелчком мышью. • `TwoClick`: Элемент активизируется двукратным щелчком мышью.
`Alignment`	Чтение/запись	Позволяет управлять тем, каким образом располагаются элементы при просмотре списков. Существую четыре возможных значения: • `Default`: Если пользователь переносит мышью некоторый элемент, то этот элемент останется там, где его "отпустит" пользователь. • `Left`: Элементы выравниваются по левому краю управляющего элемента `ListView`. • `Top`: Элементы выравниваются по верхнему краю управляющего элемента `ListView`. • `SnapToGrid`: В этом случае управляющий элемент `ListView` содержит невидимую сетку, в ячейки которой помещаются элементы.
`AllowColumnReader`	Чтение/запись	Если присвоить этому свойству значение `true`, то у пользователя появится возможность изменять относительное расположение столбцов при просмотре списков. Но в этом случае, вы должны быть уверены в том, что процедуры, ответственные за наполнение просматриваемого списка, окажутся в состоянии включать элементы надлежащим образом даже тогда, когда порядок расположения столбцов будет изменен.
`AutoArrange`	Чтение/запись	Если присвоить этому свойству значение `true`, то элементы будут автоматически выравниваться в соответствии с тем, как определено свойством `Alignment`. Если пользователь

Продолжение таблицы

Имя	Режим доступа	Описание
`AutoArrange` (*продолжение*)		попытается перенести мышью элемент в центр просматриваемого списка, а значение свойстаа `Alignment` равно `Left`, то элемент автоматически вериется к левому краю. Это свойство имеет смысл только в том случае, если значение свойства `View` равно `LargeIcon` или `SmallIcon`.
`CheckBoxes`	Чтение/запись	Если присвоить этому свойству значение `true`, то у каждого элемента просматриваемого списка появится свой `CheckBox`, который будет выведен с левой от него стороны. Это свойство имеет смысл только в том случае, если свойство `View` имеет значение `Details` или `List`.
`CheckedIndices` `CheckedItems`	Только чтение	Эти два свойства обеспечивают осуществление доступа к семействам индексов и элементов соответственно, в которых содержатся помеченные элементы списка.
`Columns`	Только чтение	Просматриваемый список может состоять из столбцов. Это свойство обеспечивает доступ к семейству столбцов, с помощью которого можно добавлять или удалять столбцы.
`FocusedItem`	Только чтение	Содержит в себе элемент просматриваемого списка, на котором в настоящий момент находится фокус. Если вообще ничего не выбрано, то его значение равно `null`.
`FullRowSelect`	Чтение/запись	Если значение этого свойства равно `true`, то при щелчке мышью на некотором элементе, будет выделяться вся строка, в котором он расположен. Если значение этого свойства равно `false`, то выделяется только сам элемент.
`GridLines`	Чтение/запись	Если значение этого свойства равно `true`, то элемент, управляющий просмотром списков, выведет сетку между рядами и столбцами. Это свойство имеет значение только в том случае, если значение свойства `View` равно `Details`.
`HeaderStyle`	Чтение/запись	Позволяет управлять тем, каким образом выводятся заголовки столбцов. Существует три различных стиля: • `Clickable`: Заголовок столбца выступает в роли кнопки. • `NonClickable`: Заголовки столбцов не реагируют на щелчок мышью. • `None`: Заголовки столбцов не выводятся.
`HoverSelection`	Чтение/запись	Если значение этого свойства равно `true`, то можно выбрать элемент, просто наведя на него курсор мыши.
`Items`	Только чтение	Семейство элементов просматриваемого списка.
`LabelEdit`	Чтение/запись	Если значение этого свойства равно `true`, то у пользователя имеется возможность редактировать содержимое первого столбца при установленном режиме просмотра `Details`.
`LabelWrap`	Чтение/запись	Если это значение этого свойства равно `true`, то надписи будут занимать столько строк, сколько необходимо для вывода всего текста.
`LargeImageList`	Чтение/запись	Содержит список изображений `ImageList`, в котором находятся большие изображения. Эти изображения могут быть использованы в том случае, когда значение свойства `View` имеет значение `LargeIcon`.
`MultiSelect`	Чтение/запись	Этому свойству следует присваивать значение `true` в тех случаях, когда необходимо разрешить пользователю выбор сразу нескольких элементов.

Продолжение таблицы

Имя	Режим доступа	Описание
`Scrollable`	Чтение/запись	Этому свойству следует присваивать значение `true`, когда необходимо выводить на экран линейки прокрутки.
`SelectedIndices` `SelectedItems`	Только чтение	В этих двух свойствах содержатся семейства, в которых хранятся выбранные индексы и элементы соответственно.
`SmallImageList`	Только чтение	Когда свойство `View` имеет значение `SmallIcon`, в этом свойстве содержится список изображений `ImageList`, в котором хранятся используемые рисунки.
`Sorting`	Чтение/запись	Можно разрешить элементу, управляющему просмотром списка, осуществлять сортировку содержащихся в нем элементов. Существует три различных режима сортировки: • `Ascending` (по возрастанию) • `Descending` (по убыванию) • `None` (никакого)
`StateImageList`	Чтение/запись	В `ImageList` хранятся маски рисунков, которые используются в качестве оверлеев для рисунков из `LargeImageList` и `SmallImageList`, отображающих изменяющиеся состояния.
`TopItem`	Только чтение	Возвращает элемент, располагающийся в самом верху просматриваемого списка.
`View`	Чтение/запись	Элемент, управляющий просмотром списка, может выводить элементы в четырех различных режимах: • `LargeIcon`: Все элементы выводятся с большой иконкой (32 × 32) и надписью. • `SmallIcon`: Все элементы выводятся с маленькой иконкой (16 ×16) и надписью. • `List`: Выводится только один столбец. У этого столбца могут быть свои иконка и надпись. • `Details`: Может выводиться произвольное число столбцов. Иконкой обладает только первый столбец.

Методы управляющего элемента ListView

Для такого сложного управляющего элемента, как `ListView` существует на удивление немного методов, относящихся исключительно только к нему. Все они описаны в приведенной ниже таблице:

Имя	Описание
`BeginUpdate`	Обратившись к этому методу, можно добиться того, что управляющий элемент `ListView` прекратит отображать изменения до тех пор, пока не будет вызван метод `EndUpdate`. Это может оказаться полезным при необходимости вставить большое количество элементов, поскольку позволяет предотвратить мерцание изображения и существенно увеличить скорость.
`Clear`	Полностью очищает элемент `ListView`. Удаляются все элементы и столбцы.
`EndUpdate`	Этот метод следует вызывать после вызова метода `BeginUpdate`. После обращения к этому методу элемент `ListView` выведет все свои элементы.
`EnsureVisible`	При обращении к этому методу управляющий элемент `ListView` сам осуществит прокрутку, с тем чтобы элемент с заданным индексом оказался видимым на экране.
`GetItemAt`	Возвращает элемент, располагающийся в координатах x, y в просматриваемом списке.

События управляющего элемента ListView

В следующей таблице приведены события управляющего элемента ListView.

Имя	Описание
`AfterLabelEdit`	Наступает после окончания редактирования надписи.
`BeforeLabelEdit`	Наступает перед тем, как пользователь приступает к редактированию надписи.
`ColumnClick`	Наступает при щелчке мышью на столбце.
`ItemActivate`	Наступает при активизации элемента.

ListViewItem

Элемент просматриваемого списка всегда представляет собой экземпляр класса `ListViewItem`. В этом классе содержится различная информация, например, текст и индекс иконки, которую необходимо вывести. Экземпляры названного класса обладают семейством с именем `SubItems`, в котором хранятся экземпляры другого класса — `ListViewSubItem`. Эти субэлементы выводятся, если управляющий элемент `ListView` находится в режиме `Details`. Каждый из субэлементов представляет собой столбец просматриваемого списка. Основные отличие субэлементов от основных элементов заключается в том, что субэлемент не может выводить иконку.

Элементы `ListViewItem` добавляются в управляющий элемент `ListView` с помощью семейства `Items`, а субэлементы `ListViewSubItem` добавляются в класс `ListViewItem` с помощью семейства `SubItems` класса `ListViewItem`.

ColumnHeader

Для того чтобы при просмотре списка выводились заголовки столбцов, необходимо добавить в семейство `Columns` управляющего элемента `ListView` экземпляры класса, называемого `ColumnHeader`. Экземпляры этого класса содержат в себе заголовки столбцов, которые могут быть выведены, если управляющий элемент `ListView` находится в режиме `Details`.

Управляющий элемент ImageList

Управляющий элемент `ImageList` предоставляет семейство, которое может быть использовано для хранения рисунков, используемых другими управляющими элементами данной формы. В этом списке рисунков могут храниться изображения произвольных размеров, однако в рамках каждого конкретного управляющего элемента все рисунки должны иметь одинаковые размеры. В случае управляющего элемента `ListView` следует использовать два управляющих элемента `ImageList` — это дает возможность выводить как большие, так и маленькие рисунки.

Управляющий элемент `ImageList` никак не отображается на экране во время выполнения. Когда вы переносите его в разрабатываемую вами форму, он размещается не на самой форме, а под ней — на поддоне, содержащем все компоненты такого рода. Эта замечательная возможность, которая позволяет не засорять разрабатываемые формы управляющими элементами, не являющимися частью пользовательского интерфейса. Этот управляющий элемент допускает выполнение над

собой абсолютно тех же самых операций, что и все остальные за исключением того, что его нельзя перемещать по форме.

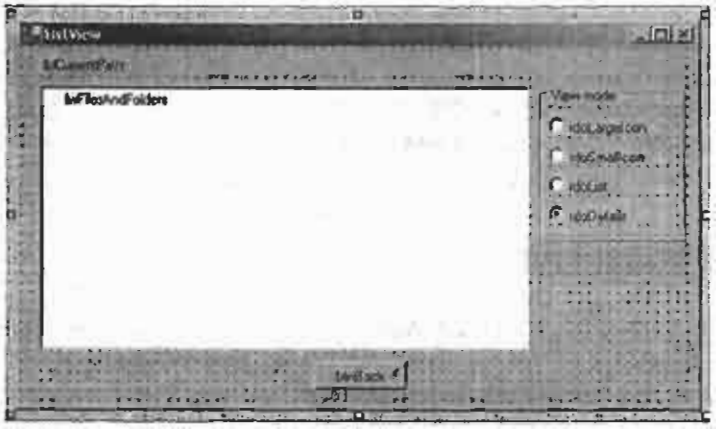
Добавлять рисунки в управляющий элемент `ImageList` можно в процессе как разработки, так и выполнения. Если в процессе разработки известно, какие рисунки будет необходимо выводить, то можно сразу добавить эти рисунки, щелкнув мышью на кнопке, расположенной с правой стороны от свойства `Images`. Это приведет к появлению диалогового окна, которое позволит добраться до рисуиков, которые необходимо включить. Если требуется добавлять рисунки в процессе выполнения, то для этого следует использовать семейство `Images`.

Позиакомиться с тем, каким образом следует применять этот управляющий элемент, можно с помощью примера, посвященного использованию управляющего элемента `ListView`.

Практикум: пример использования управляющего элемента ListView

Наилучший способ разобраться с тем, каким образом может использоваться управляющий элемент `ListView` и соответствующие ему списки рисунков,— это рассмотреть соответствующий пример. Мы создадим диалог, в котором будут использованы управляющий элемент `ListView` и два управляющих элемента `ImageList`. Управляющий элемент `ListView` будет выводить файлы и папки, расположенные на вашем жестком диске. Для простоты мы не будем извлекать настоящие икоики нз всех файлов и папок, вместо этого будем нспользовать стандартные иконки папок н иифорлационные иконки для файлов.

Щелкнув два раза мышью на папке, вы сможете перейти в дерево данной папки; мы предусмотрим также кнопку возврата, которая позволнт двигаться вверх по дереву. Для изменения режима просмотра списка в процессе выполнения используются четыре переключателя. Если на каком-либо файле пронзводится двойной щелчок мышью, то мы предпрнимаем попытку выполнення этот файл.



Как всегда, начнем с создания пользовательского нитерфейса:

1. Создайте новый проект Visual Studio.NET и назовите его `Listview`. Добавьте в форму управляющий элемент `ListView`, кнопку, надпись и групповое окио, в которое добавьте четыре переключателя — вы получите форму, изображенную на рисунке слева.
2. Присвойте управляющим элементам имена, как показано на рисунке. Управляющий элемент `ListView` не будет выводить свое имя как изображено на этом рисунке; автор добавил в него элемент с тем, чтобы присвоить ему имя. Вам так поступать не следует.
3. Измените свойство `Text` переключателей н обычной кнопки в соответствин с нх именем за исключением первых трех символов.
4. Очистите свойство `Text` надписи.

5. Добавьте в форму два управляющих элемента `ImageList` с помощью двойного щелчка мышью на иконке управляющего элемента в окне инструментов Toolbox (вам придется прокрутить его вниз, для того чтобы найти нужную иконку). Измените их имена на `ilSmall` и `ilLarge`.

6. Измените свойство `Size` управляющего элемента `ImageList` с именем `ilLarge` на 32,32.

7. Щелкните мышью на кнопке, расположенной справа от свойства `Images` управляющего элемента `ilLarge`, для перехода в диалоговое окно, позволяющее осуществлять поиск рисунков, которые вы желаете включить.

8. Щелкните мышью на Add и перейдите к папке, расположенной под Visual Studio.NET, в которой содержатся рисунки. Они содержатся в следующих файлах:

```
<Drive>:\Program Files\Visual
Studio.NET\Common7\Graphics\Icons\Win95\clsdfold.ico
```

и

```
Drive>:\Program Files\Visual
Studio.NET\Common7\Graphics\Icons\Computer\msgbox04.ico
```

9. Убедитесь в том, что иконка папки располагается на самом верху списка.

10. Повторите шаги 7 и 8 со вторым управляющим элементом `ImageList` — `ilSmall`.

11. Присвойте свойству `Checked` переключателя `rdoDetails` значение `true`.

12. Задайте следующие значения свойствам управляющего элемента просмотра списков:

```
Multiset = true
LargeImageList = ilLarge
SmallImageList = ilSmall
View = Details
```

13. Измените свойство `Text` формы и Frame, как показано на рисунке, приведенном выше.

Добавление обработчиков событий

На этом создание пользовательского интерфейса завершено, и можно переходить к созданию кода. Прежде всего потребуется поле для хранения папок, по которым осуществляется поиск, это даст возможность возвращаться к ним при нажатии кнопки возврата. Мы собираемся хранить абсолютный путь к папкам, поэтому для выполнения этой работы выбрано семейство `StringCollection`:

```
public class Form1 : System.Windows.Forms.Form
{
    // Поле, предназначенное для хранения предыдущих папок
    private System.Collections.Specialized.StringCollection folderCol;
```

При разработке формы Validated мы не создавали никаких заголовков для столбцов, поэтому это придется сделать сейчас. Они создаются с использованием метода CreateHeadersAndFillListView():

```
private void CreateHeadersAndFillListView()
{
   ColumnHeader colHead;

   // Первый заголовок
   colHead = new ColumnHeader();
   colHead.Text = "Filename";
   this.lwFilesAndFolders.Columns.Add(colHead);     // Вставьте заголовок

   // Второй заголовок
   colHead = new ColumnHeader();
   colHead.Text = "Size";
   this.lwFilesAndFolders.Columns.Add(colHead);     // Вставьте заголовок

   // Третий заголовок
   colHead = new ColumnHeader();
   colHead.Text = "Last accessed";
   this.lwFilesAndFolders.Columns.Add(colHead);     // Вставьте заголовок
}
```

Для начала мы объявляем одну переменную — ColHead, которая будет использоваться для создания трех заголовков столбцов. Для каждого из них мы обновляем эту переменную и присваиваем ей Text перед тем, как добавить ее к семейству Columns управляющего элемента ListView.

Для завершения инициализации формы при ее первом выводе остается заполнить просматриваемый список именами файлов и папок, хранящихся на жестком диске. Это достигается за счет использования еще одного метода:

```
private void PaintListView(string root)
{
   try
   {
      // Две локальные переменные, которые используются при создании элементов,
      // подлежащих включению
      ListViewItem lvi;
      ListViewItem.ListViewSubItem lvsi;

      // Если отсутствует корневая папка, то мы не имеем возможности включать
      // что бы то ни было
      if (root.CompareTo("") == 0)
         return;

      // Получение информации о корневой папке.
      System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(root);

      // Извлечение файлов и папок, находящихся в корневой папке
      DirectoryInfo[] dirs = dir.GetDirectories();      // Папки
      FileInfo[] files = dir.GetFiles();                // Файлы

      // Очистка ListView. Обратите внимание на то, что вызывается метод Clear
      // семейства Items, а не самого управляющего элемента ListView. Метод Clear
      // управляющего элемента ListView удаляет все, включая заголовки столбцов,
      // а нам необходимо удалить только элементы из просматриваемого списка.
      this.lwFilesAndFolders.Items.Clear();

      // Маркировка текущего пути
      this.lblCurrentPath.Text = root;
```

```
            // Запрет на внесение каких бы то ни было изменений в ListView
            this.lwFilesAndFolders.BeginUpdate();

            // Проход в цикле по всем папкам, расположенным в корневой папке,
            // и включение их в список
            foreach (System.IO.DirectoryInfo di in dirs)
            {
               // Создание основного элемента ListViewItem
               lvi = new ListViewItem();
               lvi.Text = di.Name;              // Имя папки
               lvi.ImageIndex = 0;              // Индекс иконки данной папки равен 0
               lvi.Tag = di.FullName;           // Присваивание тегу заданного пути к файлу

               // Создание двух элементов ListViewSubItems.
               lvsi = new ListViewItem.ListViewSubItem();
               lvsi.Text = "";                  // Размер - у папок размер отсутствует, поэтому
                                                // данный столбец остается пустым
               lvi.SubItems.Add(lvsi);          // Включение субэлемента в элемент ListViewItem
               lvsi = new ListViewItem.ListViewSubItem();
               lvsi.Text = di.LastAccessTime.ToString();     // Последний столбец, с которым
                                                             // велась работа
               lvi.SubItems.Add(lvsi);          // Включение субэлемента в элемент ListViewItem

               // Включение элемента ListViewItem в семейство Items управляющего
               // элемента ListView
               this.lwFilesAndFolders.Items.Add(lvi);
            }

            // Проход в цикле по всем файлам, находящимся в корневой папке
            foreach (System.IO.FileInfo fi in files)
            {
               // Создание основного элемента ListViewItem
               lvi = new ListViewItem();
               lvi.Text = fi.Name;              // Filename
               lvi.ImageIndex = 1;              // Иконка, которая используется для представления
                                                // папки имеет индекс, равный 1
               lvi.Tag = fi.FullName;           // Присваивание тегу заданного пути к файлу

               // Создание двух субэлементов
               lvsi = new ListViewItem.ListViewSubItem();
               lvsi.Text = fi.Length.ToString();       // Длина файла
               lvi.SubItems.Add(lvsi);                 // Добавление в семейство SubItems

               lvsi = new ListViewItem.ListViewSubItem();
               lvsi.Text = fi.LastAccessTime.ToString();     // Последний столбец, с которым
                                                             // велась работа
               lvi.SubItems.Add(lvsi);                 // Добавление в семейство SubItems

               // Добавление элемента в семейство Items управляющего элемента ListView
               this.lwFilesAndFolders.Items.Add(lvi);
            }

            // Снятие запрета на внесение изменений в ListView. Теперь элементы,
            // которые были включены, будут выведены на экран
            this.lwFilesAndFolders.EndUpdate();
         }
         Catch (System.Exception err)
         {
            MessageBox.Show("Error: " + err.Message);
         }
      }
```

До того как начинает выполняться первый из двух блоков `foreach`, мы обращаемся к методу `BeginUpdate()` управляющего элемента `ListView`. Этот метод сообщает о том, что необходимо прекратить вносить изменения в видимую часть окна до тех пор, пока не будет вызван метод `EndUpdate()`. Если не обратиться к этому методу, заполненне просматриваемого списка будет происходить медленнее, а сам список будет мелькать при добавлении новых элементов. Только после выполнения второго блока `foreach` мы обращаемся к методу `EndUpdate()`, в результате чего управляющий элемент `ListView` выводит на экран все элементы, которые мы ему передали.

Интересующий нас код содержится в этих двух блоках `foreach`. Мы начннаем с того, что создаем новый экземпляр `ListViewItem`, а затем прнсваиваем его свойству `Text` имя того файла или той папкн, которые собираемся вставлять. Значение свойства `ImageIndex` экземпляра `ListViewItem` указывает на индекс элемента, расположенного в одном из двух спнсков `ImageList`. Именно по этой прнчине представляется важным, чтобы соответствующие рисунки имели одинаковые индексы в обоих списках `ImageList`. Для хранения полностью квалнфицированного пути как к папкам, так и к файлам, который может понадобиться в том случае, если пользователь щелкнет на данном элементе мышью два раза, нспользуется свойство `Tag`.

Далее мы используем два субэлемента. Им просто прнсванвается текст, который необходимо вывести, после чего онн добавляются в семейство `SubItems` экземпляра `ListViewItem`.

В завершение экземпляр `ListViewItem` добавляется в семейство `Items` элемента `ListView`. Управляющий элемент `ListView` оказывается достаточно интеллектуальным, чтобы игнорировать любые субэлементы, еслн используется какой-лнбо режнм вывода, отличный от `Details`, поэтому субэлементы добавляются независимо от того, какой именно режим установлен в настоящий момент.

Все, что остается сделать, чтобы включить в просматриваемый список корневую папку,— это вызвать две функции в конструкторе формы. Одновременно мы создаем экземпляр `folderCol` семейства `StringCollection` с корневой папкой:

```
InitializeComponent();

// Инициализация ListView и семейства папок
folderCol = new System.Collections.Specialized.StringCollection();
CreateHeadersAndFillListView();
PaintListView(@"C:\");
folderCol.Add(@"C:\");
```

Чтобы иметь возможность перемещаться по папкам, изображенным в управляющем элементе `ListView`, посредством двойного щелчка на папке мышью, необходимо подписаться на событие `ItemActivate`. Это также выполняется в конструкторе:

```
this.lwFilesAndFolders.ItemActivate += new
                System.EventHandler(this.lwFilesAndFolders_ItemActivate);
```

Обработчик соответствующего события выглядит следующим образом:

```
private void lwFilesAndFolders_ItemActivate(object sender, System.EventArgs e)
{
  // Приведение типа отправляемого объекта к типу ListView и получение тега
  // первого выбранного элемента.
  System.Windows.Forms.ListView lw = (System.Windows.Forms.ListView)sender;
  string filename = lw.SelectedItems[0].Tag.ToString();

  if (lw.SelectedItems[0].ImageIndex != 0)
  {
    try
```

```
        {
            // Попытка запустить файл.
            System.Diagnostics.Process.Start(filename);
        }
        catch
        {
            // Если попытка оканчивается неудачей, то мы просто выходим
            // из данного метода.
            return;
        }
    }
    else
    {
        // Включение элементов
        PaintListView(filename);
        folderCol.Add(filename);
    }
}
```

Свойство `Tag` выбранного элемента содержит полностью заданный путь к тому файлу (папке), на котором был произведен двойной щелчок мышью. Известно, что рисунок с индексом 0 представляет собой папку, поэтому с помощью этого индекса можно определить, является ли данный объект файлом или папкой. Если этот объект — файл, то предпринимается попытка загрузить его.

Если этот объект является папкой, то вызывается метод `PaintListView` с именем новой папки, а затем добавляется новая папка в семейство `folderCol`.

Перед тем как перейти к переключателям, расширим наши поисковые возможности, добавив событие "щелчок мышью" для кнопки Back. Щелкните на этой кнопке мышью два раза и вне·ите в обработчик события следующий код:

```
private void btnBack_Click(object sender, System.EventArgs e)
{
    if (folderCol.Count > 1)
    {
        PaintListView(folderCol[folderCol.Count-2].ToString());
        folderCol.RemoveAt(folderCol.Count-1);
    }
    else
    {
        PaintListView(folderCol[0].ToString());
    }
}
```

Если семейство `folderCol` состоит более, чем из одного элемента, то это значит, что мы находимся не в корне просмотра, и мы вызываем метод `PaintListView` с указанием пути к предшествующей папке. Последним элементом в семействе является текущая папка, поэтому нам потребуется второй последний элемент. Затем мы удаляем последний элемент семейства и делаем текущим тот новый элемент, который стал последним в результате этого. Если в семействе имеется всего один элемент, то мы просто вызываем метод `PaintListView` с этим элементом.

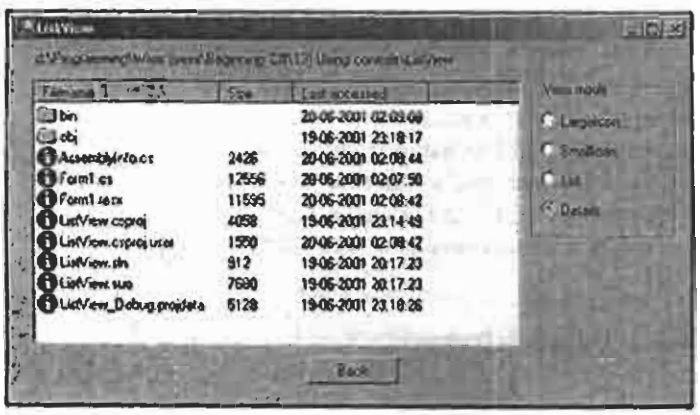
Все, что нам теперь осталось предусмотреть, — это возможность изменять тип просмотра списка. Щелкните мышью два раза на каждом переключателе и добавьте следующий код:

```
private void rdoLarge_CheckedChanged(object sender, System.EventArgs e)
{
    RadioButton rdb = (RadioButton)sender;
    if (rdb.Checked)
        this.lwFilesAndFolders.View = View.LargeIcon;
}
```

```
private void rdoList_CheckedChanged(object sender, System.EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
      this.lwFilesAndFolders.View = View.List;
}

private void rdoSmall_CheckedChanged(object sender, System.EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
      this.lwFilesAndFolders.View = View.SmallIcon;
}

private void rdoDetails_CheckedChanged(object sender, System.EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
      this.lwFilesAndFolders.View = View.Details;
}
```



Мы осуществляем проверку переключателя на предмет того, не изменилось ли значение его свойства `Checked` на `true`, и если изменилось, то мы присваиваем соответствующее значение свойству `View` управляющего элемента `ListView`.

На этом пример использования управляющего элемента `ListView` завершен. Запустив его, вы сможете увидеть приблизительно то, что показано на рисунке слева.

Управляющий элемент StatusBar

Управляющий элемент `StatusBar` (линейка состояния) обычно используется для предоставления дополнительных данных о выбранном элементе или информации о действии, выполняемом в настоящий момент в рамках диалога. Обычно линейка состояния располагается у нижней границы экрана, как, например, в приложениях MS Office, хотя ее можно расположить и в любом другом месте. Линейка состояния, поставляемая вместе с Visual Studio.NET, может использоваться просто для вывода текста, либо вы можете добавить в нее собственные панели или создать собственные программы вывода содержимого панели на экран:



На рисунке изображена панель состояния в том виде, в котором она используется в MS Word. Панели в линейке состояния могут быть определены как слегка притопленные участки.

Свойства управляющего элемента StatusBar

Существует возможность присвоить свойству `Text` управляющего элемента `StatusBar` некоторый текст, который будет выведен пользователю; кроме этого можно создавать панели и использовать их для тех же целей:

Имя	Режим доступа	Описание
`BackGroundImage`	Чтение/запись	К линейке состояния можно привязать какой-либо рисунок, который будет выводиться на заднем плане.
`Panels`	Только чтение	Представляет собой семейство панелей, из которых состоит линейка состояния. Это семейство используется для добавления и удаления панелей.
`ShowPanels`	Чтение/запись	Этому свойству должно быть присвоено значение `true` в том случае, если будут выводиться панели.
`Text`	Чтение/запись	В том случае, если панели не используются, в этом свойстве содержится текст, который выводится в линейке состояния.

События управляющего элемента StatusBar

Вряд ли можно ожидать большого количества новых событий для линейки состояния, но если вы рисуете панель вручную, то событие `DrawItem` будет представлять особую важность:

Имя	Описание
`DrawItem`	Событие наступает, когда нужно заново нарисовать панель, обладающую набором стилей `OwnerDraw`. На это событие необходимо подписаться, если вы намерены рисовать содержимое панели самостоятельно.
`PanelClick`	Событие наступает в тот момент, когда производится щелчок мышью на панелях.

Класс StatusBarPanel

Каждая панель в линейке состояния представляет собой экземпляр класса `StatusBarPanel`. В этом классе содержится исчерпывающая информация обо всех панелях семейства `Panels`. Информация, которая может задаваться, варьируется от простого текста и его местоположения до выводимых иконок и стиля панели.

Если возникает необходимость нарисовать панель самостоятельно, то следует присвоить свойству `Style` значение `OwnerDraw` и работать с событием линейки состояния `DrawItem`.

Практикум: пример использования управляющего элемента StatusBar

Для демонстрации управляющего элемента `StatusBar` внесем соответствующие изменения в созданный ранее пример использования элемента `ListView`. Мы удалим надпись, использовавшуюся для вывода текущей папки, и будем отображать эту информацию на панели, расположенной в линейке состояния. Мы также выведем вторую панель, на которой будет отображаться текущий режим просмотра элемента `ListView`:

1. Удалите надпись `lblCurrentFolder`.

2. Щелкните два раза мышью на элементе `StatusBar` в панели инструментов, чтобы добавить его в форму (оно будет расположено около нижней границы списка). Новый управляющий элемент автоматически будет прижат к нижнему краю формы.

3. Измените имя элемента `StatusBar` на `sbInfo` и очистите свойство `Text`.

4. Найдите свойство `Panels` и два раза щелкните мышью на кнопке, расположенной справа от него, для перехода в диалог, который позволяет добавлять панели.

5. Щелкните мышью на Add — в семейство будет добавлена новая панель. Присвойте свойству `AutoSize` значение `Spring`. Это означает, что данная панель разместится в элементе `StatusBar` вместе с другими панелями.

6. Щелкните мышью на Add еще раз и присвойте свойству `AutoSize` значение `Contents`. Это означает, что размер панели будет автоматически изменяться в соответствии с размерами содержащегося в ней текста. Присвойте свойству `MinSize` значение 0.

7. Нажмите OK для закрытия диалога.

8. Присвойте свойству `ShowPanels` элемента `StatusBar` значение `true`.

Этот пример можно найти в коде проекта `StatusBar` Visual Studio.NET.

Добавление обработчиков событий

На этом разработка пользовательского интерфейса завершена, и мы переходим к созданию кода. Начнем с задания текущего пути с помощью метода `PaintListView`. Удалите строку, в которой текст присваивается надписи, и вставьте на ее место следующее:

```
this.sbInfo.Panels[0].Text = root;
```

Самая первая панель имеет индекс 0, и в этой строке мы просто присваиваем ее свойство `Text` — точно так же, как мы задавали свойство `Text` надписи. В завершение вносим изменения в события для четырех переключателей, чтобы задать вторую панель:

```
private void rdoLarge_CheckedChanged(object sender, System.EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
   {
      this.lwFilesAndFolders.View = View.LargeIcon;
      this.sbInfo.Panels[1].Text = "Large Icon";
   }
}

private void rdoList_CheckedChanged(object sender, System.EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
   {
      this.lwFilesAndFolders.View = View.List;
```

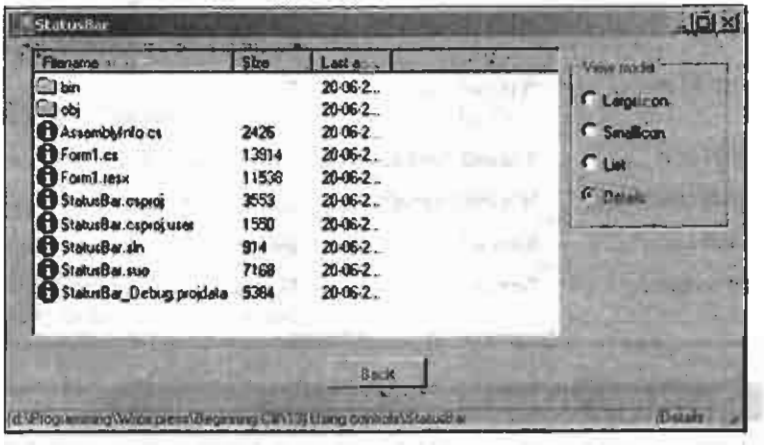
```
            this.sbInfo.Panels[1].Text = "List";
        }
    }

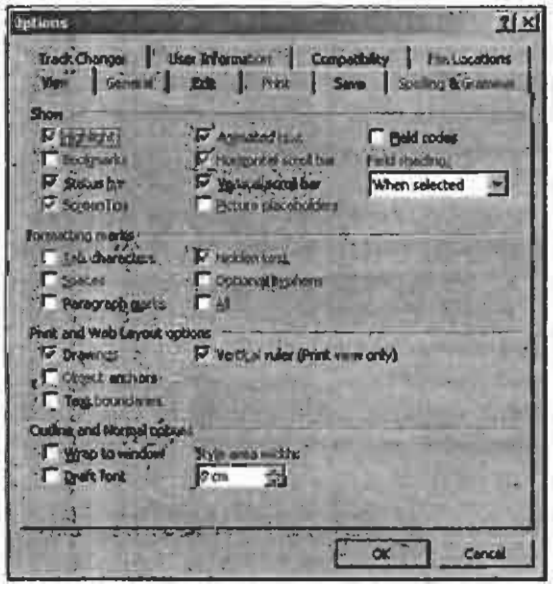
    private void rdoSmall_CheckedChanged(object sender, System.EventArgs e)
    {
        RadioButton rdb = (RadioButton)sender;
        if (rdb.Checked)
        {
            this.lwFilesAndFolders.View = View.SmallIcon;
            this.sbInfo.Panels[1].Text = "Small Icon";
        }
    }

    private void rdoDetails_CheckedChanged(object sender, System.EventArgs e)
    {
        RadioButton rdb = (RadioButton)sender;
        if (rdb.Checked)
        {
            this.lwFilesAndFolders.View = View.Details;
            this.sbInfo.Panels[1].Text = "Details";
        }
    }
```

Текст панели в подобном случае задается способом, аналогичным приведенному выше с использованием `PaintListView`.

На этом пример использования управляющего элемента `StatusBar` завершен. Если вы его запустите, то увидите на экране что-то, подобное изображенному на рисунке справа.





Управляющий элемент TabControl

Элемент `TabControl` предоставляет простой способ структурирования диалога в виде отдельных логических частей, доступ к которым осуществляется с помощью вкладок, располагаемых в самой верхней части управляющего элемента. `TabControl` обладает свойством `TabPages`, которое работает аналогично управляющему элементу `GroupBox`, хотя и является несколько более сложным.

На экране, изображенном на рисунке слева, показан диалог Options из MS Word 2000, поскольку его конфигурация является типичной. Обратите внимание на два ряда вкладок в верхней части диалога. Щелчок мышью на каждой из них будет приводить к выводу различных наборов управляющих

элементов для последующего диалога. Это очень хороший пример того, каким образом можно использовать вкладки для разбиения информации на логически связанные группы, что значительно упрощает нахождение необходимой информации.

Использовать элемент `TabControl` весьма просто. Необходимо просто задать число вкладок в семействе `TabPages` данного элемента, а затем перенести мышью те управляющие элементы, которые должны располагаться в соответствующих страницах диалога.

В процессе выполнения программы перемещение между вкладками осуществляется с помощью свойств этого управляющего элемента.

Свойства элемента TabControl

Имя	Режим доступа	Описание
`Alignment`	Чтение/запись	Определяет, в каком месте элемента `TabControl` выводятся вкладки. По умолчанию они выводятся наверху.
`Appearance`	Чтение/запись	Управляет тем, как изображаются вкладки. Они могут выводиться либо в виде обычных кнопок, либо плоскими.
`HotTrack`	Чтение/запись	Если значение этого свойства равно `true`, то внешний вид вкладок изменяется по мере того, как на них наводится курсор мыши.
`MultiLine`	Чтение/запись	Если значение этого свойства равно `true`, то допускается расположение вкладок в несколько рядов.
`RowCount`	Только чтение	Возвращает число рядов вкладок, выведенных в настоящий момент.
`SelectedIndex`	Чтение/запись	Позволяет получить или задать индекс выбранной вкладки.
`TabCount`	Только чтение	Возвращает общее число вкладок.
`TabPages`	Только чтение	Представляет собой семейство страниц `TabPages` данного элемента. Используется для добавления и удаления страниц.

Практикум: работа с управляющим элементом TabControl

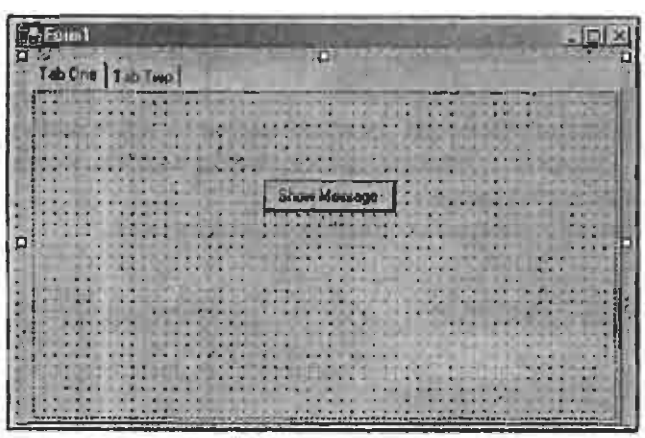
Элемент `TabControl` работает несколько иначе, чем встретившиеся нам другие управляющие элементы. Когда вы перенесете мышью этот элемент в форму, то увидите серый прямоугольник, который совсем не похож на то предыдущие изображения управляющих элементов. Вы также обнаружите ниже панели Properties две кнопки с названиями Add Tab и Remove Tab. Щелчок мышью на Add Tab приведет к добавлению новой вкладки в данный элемент, и он примет привычный вид. Очевидно, что с помощью кнопки Remove Tab можно удалить уже существующую вкладку.

Эта процедура предназначена для упрощения знакомства и быстрого использования данного управляющего элемента. Если же, напротив, вам требуется внести изменения в работу или стиль вкладок, то в таком случае необходимо воспользоваться диалогом `TabPages`; доступ к нему можно получить с помощью специальной кнопки, которая появляется, когда вы выбираете `TabPages` в панели свойств.

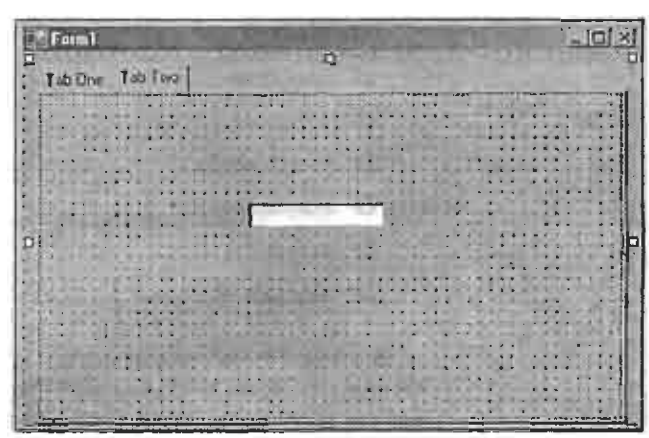
Свойство `TabPages` является также тем семейством, которое используется для доступа к отдельным страницам элемента, управляющего вкладками. Создадим пример, иллюстрирующий, каким образом можно получить доступ к управляющим элементам, расположенным на разных страницах:

1. Создайте новый проект C# Windows Application и назовите его `TabControl`.

2. Перенесите управляющий элемент `TabControl` из панели инструментов в форму.
3. Щелкните мышью на Add Tab, для того чтобы добавить вкладку к управляющему элементу.
4. Найдите свойство `TabPages` и, выбрав его, щелкните мышью на кнопке, расположенной с правой стороны.
5. Нажатием кнопки Add Tab добавьте в управляющий элемент еще одну страницу с вкладкой.
6. Измените свойство `Text` страницы с вкладкой на Tab One и Tab Two соответственно.
7. Теперь вы можете продолжить работу с этими страницами, выбирая необходимую вам с помощью соответствующей вкладки в верхней части управляющего элемента. Выберите вкладку с текстом Tab One. Перенесите на нее кнопку. Убедитесь, что кнопка размещена внутри рамки `TabControl`. Если кнопка окажется вне границ, то она будет относиться к форме, а не к данному элементу.
8. Измените название кнопки на `btnShowMessage`, а свойство `Text` кнопки на Show Message.



9. Щелкните мышью на вкладке, свойство `Text` которой имеет значение Tab Two. Перенесите мышью управляющий элемент `Textbox` на поверхность элемента `TabControl`. Назовите этот управляющий элемент `txtMessage` и очистите его свойство `Text`.
10. Нажмите в диалоге OK и вернитесь к форме. Две вкладки должны иметь вид такой, как на рисунках слева и ниже.



Добавление обработчика событий

Теперь мы готовы к тому, чтобы работать с этими управляющими элементами. Если запустить код в существующем виде, то появятся страницы с вкладками. Все, что остается сделать для демонстрации элемента, управляющего вкладками,— это добавить некоторый код, который при нажатии кнопки Show Message выводит в окошке сообщения текст, введенный на другой вкладке. Прежде всего, добавим обработчик события `Click` посредством двойного щелчка мышью на кнопке на первой странице с вкладкой и добавления следующего кода:

```
private void btnShowMessage_Click(object sender, System.EventArgs e)
{
    // Доступ к Textbox
    MessageBox.Show(this.txtMessage.Text);
}
```

Доступ к управляющему элементу, расположенному на странице с вкладкой, осуществляется точно так же, как к любому другому управляющему элементу, расположенному в форме. Мы считываем свойство `Text` элемента `Textbox` и выводим его в окно для сообщений.

В форме может существовать только один выбранный переключатель (кроме тех, которые находятся в групповых окнах). `TabPages` работает точно так же, как и групповые окна, что позволяет использовать множество наборов переключателей на различных страницах без необходимости использовать групповые окна.

И последнее, что необходимо знать при работе со элементом `TabControl`, это то, каким образом можно определить, какая из вкладок используется в настоящий момент. С этой целью можно воспользоваться двумя свойствами: `SelectedTab` и `SelectedIndex`. Как предполагает их названия, `SelectedTab` возвращает объект `TabPage` или `null`, если никакая вкладка не используется, а `SelectedIndex` возвращает индекс используемой вкладки или –1, если никакая вкладка не используется.

Итоги

В настоящей главе мы познакомились с некоторыми из управляющих элементов, наиболее часто используемых при создании приложений Windows, и увидели, каким образом они могут быть использованы для создания простых и одновременно мощных пользовательских интерфейсов. Мы обсудили свойства и события этих элементов и привели примеры их использования.

В данной главе были рассмотрены следующие управляющие элементы:

- `Label`
- `Button`
- `RadioButton`
- `CheckBox`
- `ComboBox`
- `ListBox`
- `ListView`
- `GroupBox`
- `RichTextbox`
- `StatusBar`
- `ImageList`
- `TabControl`

В главе 14 мы познакомимся с более сложными управляющими элементами, такими как меню и панели инструментов, а также с использованием этих элементов для разработки Multi-Document Interface (MDI, многодокументный интерфейс) для приложений Windows. Мы также продемонстрируем, каким образом можно создавать пользовательские управляющие элементы, в которых в различных сочетаниях используются простые элементы, рассмотренные в настоящей главе.

14

Дополнительные возможности Windows Form

Использование управляющих элементов, наиболее часто используемых при разработке приложений Windows, позволяет создавать весьма впечатляющие диалоговые, однако очень мало серьезных приложений Windows, которые обладают пользовательским интерфейсом, состоящим всего из одного диалогового окна. Напротив, такие приложения используют либо Single Document Interface (SDI, однодокументный интерфейс), либо Multi-Document Interface (MDI, многодокументный интерфейс). Приложения обоих этих типов обычно интенсивно используют различные меню и панели инструментов — управляющие элементы, которые еще не рассматривались.

Начнем с рассмотрения таких управляющих элементов, как "меню" и "панель инструментов". В результате мы узнаем, каким образом можно привязывать кнопки к панелям инструментов и наоборот. Затем перейдем к созданию SDI- и MDI-приложений, уделив главное внимание последним, поскольку SDI-прнложения, в основном, представляют собой подмножества MDI-приложений.

Управляющие элементы, которые поставляются в готовом виде вместе с Visual Studio.NET, являются очень мощными и предоставляют широкий спектр функциональных возможностей, однако иногда возникают ситуации, когда их использование оказывается недостаточным. Именно поэтому предусмотрена возможность создавать свои собственные управляющие элементы, и в конце данной главы мы познакомимся с тем, каким образом эта возможность реализуется.

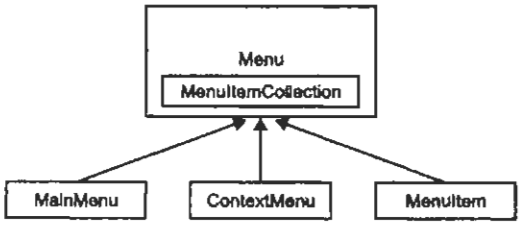
Меню

Интересно, сколько приложений Windows вы сможете вспомнить, в которых бы не было какого-нибудь меню? Вероятнее всего, что такое число будет очень близким к нулю. Меню, таким образом, можно представить себе как очень важную составную часть любого приложения, которое создается для использования под операционной системой Windows. Для облегчения работы по созданию меню для приложений в Visual Studio.NET предусмотрен управляющий элемент, который позволяет очень быстро создавать простые меню, затрачивая не намного больше усилий на создание сложных.

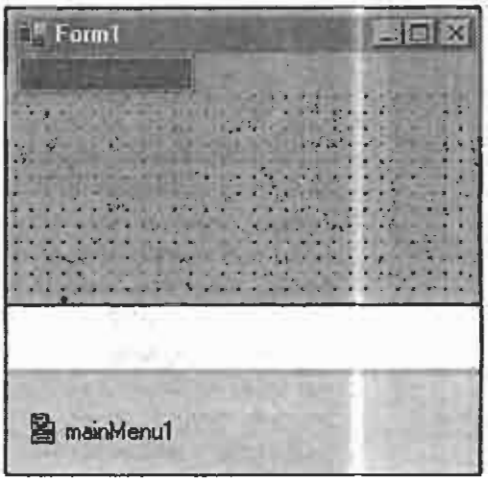
Использование управляющего элемента Menu

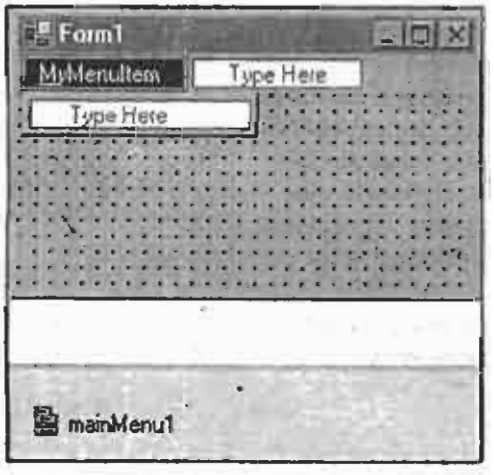
Управляющий элемент `Menu` (меню), который, который будет здесь использоваться, называется `MainMenu`. Он является, как и два других вида меню, поставляемых с Visual Studio.NET, производным от базового класса с именем `Menu`.

Второй разновидностью меню, которая будет кратко обсуждаться после `MainMenu`, является `ContextMenu`, выводимое в тот момент, когда пользователь щелкает правой кнопкой мыши на каком-либо объекте. Технически существует еще и третья разновидность меню, называемая `MenuItem` и также являющаяся производной от класса `Menu`. `MenuItem` представляет собой отдельные элементы, выводящиеся в виде меню. На рисунке слева поле Type Here (вводить здесь) как раз является примером `MenuItem`. К подробному обсуждению `MenuItem` мы перейдем очень скоро.



Если попытаться перенести управляющий элемент `MainMenu` из панели инструментов на рабочую поверхность, то можно заметить, что поведение этого элемента очень напоминает поведение панели состояния в том, что касается его одновременного размещения как на самой форме, так и под ней. Однако в отличие от панели состояния управляющий элемент меню допускает непосредственное редактирование прямо в форме. Для того чтобы создать новые пункты меню, требуется просто переместить курсор в окно с надписью Type Here (см. рис. справа).





Введите название меню в выделенное окошко, поставив амперсанд (&) перед буквой, которую вы хотите использовать в качестве мнемонического символа для данного пункта меню, то есть перед той буквой, которая будет подчеркнута в названии пункта меню и по которой к нему можно обратиться с помощью клавиш *Alt+Ключ* (см. рис. слева).

Заметьте, что вполне допустимо создать несколько пунктов меню с одним и тем же мнемоническим символом. Правило гласит, что любая буква может быть использована в каждом раскрывающемся меню только один раз (например, один раз в раскрывающемся меню Files, один раз в меню View и т. д.). Если вы случайно присвоите одну и ту же мнемоническую букву нескольким пунктам в одном раскрывающемся меню, то обнаружите, что на данный символ реагирует только самый верхний пункт.

В качестве разделителя при наборе текста в меню используется дефис (-). Это недопустимо для пунктов меню самого верхнего уровня, а может применяться только для пунктов меню более низких уровней.

Если щелкать мышью на выделенном окне, то можно заметить, что свойства, выводящиеся на панели свойств, изменились. Это происходит потому, что большинство свойств, которые могут использоваться при работе с меню, имеют отношение к отдельным пунктам меню.

Средство управления MenuItem

Если выбрать пункт меню в управляющем элементе, расположенном на форме, то можно увидеть некоторые свойства, которые используются для управления внешним видом и поведением этого пункта. Каждый пункт меню может содержать в себе другие пункты меню, что позволяет создавать подменю для каждого пункта основного меню. Для хранения всех объектов `MenuItem` в базовом классе `Menu` определено семейство `MenuItemCollection`, как показано на приведенной выше диаграмме классов.

Имя	Режим доступа	Описание
`BarBreak`	Чтение/запись	Присванвая это значение, вы указываете, что меню должно появляться с новой строки. С помощью этого свойства можно создавать меню с несколькими рядами верхнего уровня. Если это свойство задано, то пункты подменю будут располагаться в новых столбцах. Идентификатор этого свойства не выводится в Properties Window.
`Checked`	Чтение/запись	Указывает, является ли данный пункт меню выбранным.
`DefaultItem`	Чтение/запись	Пункт, выбираемый по умолчанию; рисуется жирным шрифтом. Если выполняется двойной щелчок мышью на `MenuItem`, в котором содержатся подпункты и один из них является подпунктом по умолчанию, то именно он и будет выбран.
`Enabled`	Чтение/запись	Пункт, свойству `Enabled` которого присвоено значение `false`, выводится серым цветом, и его нельзя выбрать.
`MdiList`	Чтение/запись	Указывает, будет ли в меню включен список дочерних окон в MDI.
`MergeOrder`	Чтение/запись	Определяет, каким образом будет располагаться данное меню, если оно объединяется с другим меню. Объединение различных меню будет рассматриваться ниже в настоящей главе при обсуждении MDI-приложений.
`MergeType`	Чтение/запись	Позволяет управлять поведением меню, когда его необходимо объединить с другим меню, имеющим такое же значение свойства `MergeOrder`.
`Mnemonic`	Только чтение	Возвращает мнемонический символ, поставленный в соответствие данному пункту меню. Мнемонический символ — это первый символ, следующий за амперсандом (&). Это свойство не выводится в Properties Window.
`OwnerDraw`	Чтение/запись	Если значение этого свойства равно `true`, то вся ответственность за рисунки в `MenuItem` ложится на вас. Если вам необходимо добавить в меню рисунки, то этому свойству следует присвоить значение `true` и реализовать для данного пункта обработчики для событий `DrawItem` и `MeasureItem`.
`RadioCheck`	Чтение/запись	Если это свойство имеет значение, равное `true`, то окно выбора выводится в виде переключателя, а не в виде окошка с флажком.
`Shortcut`	Чтение/запись	Это свойство определяет комбинацию клавиш, которая позволяет осуществлять непосредственный доступ к пунктам меню (например, *Ctrl+S* для пункта меню **Save**).
`ShowShortcut`	Чтение/запись	Присвоение этому свойству значения `true` будет означать, что комбинация клавиш для непосредственного доступа будет выводиться в `MenuItem` в текстовом виде.
`Text`	Чтение/запись	Собственно текст `MenuItem`. В него входит амперсанд (&) перед символом, который будет использоваться в качестве мнемонического.

Некоторые свойства MenuItem наследуются от класса Menu. Этот список не является исчерпывающим — если необходим полный перечень, следует обратиться к MSDN Reference (Справочник MSDN).

События управляющего элемента MenuItem

Сколько различных событий необходимо для MenuItem? Вы, вероятно, думаете, что достаточно одного — события Click. На самом деле, существует пять событий, на которые можно подписаться, причем два из них следует использовать, только если вы рисуете пункты самостоятельно. Ниже даны описания событий и пример, демонстрирующий использование этого управляющего элемента:

Имя	Описание
Click	Наступает, когда пользователь щелкает мышью на пункте меню. Это событие можно также сгенерировать искусственно, обратившись к методу PerformClick(), имеющемуся в MenuItem.
DrawItem	Возникает, когда пункт меню должен быть выведен. Если свойству OwnerDraw присвоено значение true, то это событие необходимо обрабатывать.
MeasureItem	Возникает перед тем, как MenuItem начинает рисоваться. Если вы рисуете MenuItem самостоятельно, то вам необходимо вычислить и вернуть ширину и высоту пункта, вызвавшего наступление события.
Popup	Возникает в момент вывода списка подпунктов данного MenuItem. Это событие может использоваться для проверки доступности пунктов меню и соответствующего задания их состояния.
Select	Наступает, когда пользователь перемещает курсор мыши на данный пункт или когда он перемещает фокус с помощью клавиатуры.

Средство управления ContextMenu



Контекстное меню — это меню, которое открывается, когда пользователь щелкает правой кнопкой мыши на каком-либо объекте, расположенном в форме. Контекстное меню, изображенное на рисунке слева, представляет собой меню, которое автор получил, щелкнув правой кнопкой мыши на рабочем столе своего компьютера

Создать контекстное меню можно посредством переноса такого меню в форму и добавления к нему необходимых пунктов — точно так же, как это может быть сделано со управляющим элементом MainMenu. Единственное отличие заключается в том, что в контекстном меню не допускается наличие пунктов самого верхнего уровня. Для того чтобы привязать контекстное меню к какому-либо управляющему элементу в форме, следует выбрать соответствующий управляющий элемент и присвоить его свойству ContextMenu указатель на созданное вами контекстное меню.

Практикум: пример использования меню

Теперь можно создать пример использования различных меню, который будет использоваться в качестве базового по мере создания более сложных примеров с использованием меню при обсуждении MDI-приложений. В данном примере мы создадим одно основное меню и одно контекстное меню:

1. Начните с создания нового Windows Application.
   Назовите его MenuExample.

2. Добавьте в форму управляющие элементы `MainMenu` и `ContextMenu` посредством двойного щелчка мышью на соответствующих управляющих элемеитах, находящихся в панели инструментов. Назовите эти управляющие элементы соответствеино `MainMenuFiles` и `ContextMenuFonts`.

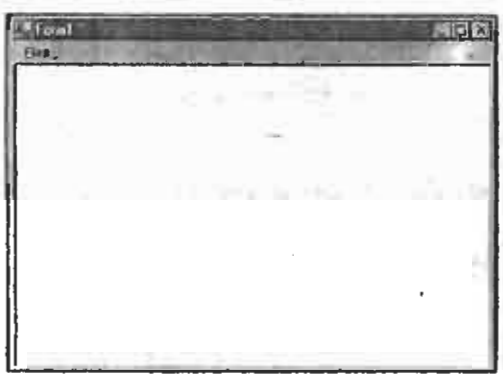
Название	Текст
menuItemFiles	&Files
menuItemNew	&New
menuItemOpen	&Open
menuItemSave	&Save
menuItemSeparator0	-
menuItemExit	E&xit

3. Добавьте приведенные ниже пункты в управляющий элемент `MainMenu`, выбрав его и вводя текст в соответствующие пункты меню, начав с `menuItemFiles` и двигаясь вниз. Меню `menuItemFiles` как меню самого верхнего уровия. Помните, что вы создаете разделитель, вводя в меню один дефис (-), а также определяете мнемонические символы для каждого пункта, включая амперсанд (&) перед символом, который выбирается в качестве мнемонического (см. табл. слева).

4. Выберите контекстное меню и создайте следующие пункты меню точно так же, как вы делали это выше (см. табл. справа).

Имя	Текст
menuItemBold	&Bold
menuItemItalic	&Italic

5. Добавьте к форме один управляющий элемент `RichTextBox`. Назовите его `rtfText` и присвойте его свойству `Dock` значение `Fill`, для того чтобы оно заполнило всю форму целиком, и очистите его свойство `Text`.



6. Выберите элемент `RichTextbox` и выберите свойство `ContextMenu`. Из раскрываемого списка выберите `ContextMenuFonts`, для того чтобы привязать контекстное меню к управляющему элементу `RichTextbox`.

После этого вы должны получить форму, которая выглядит приблизительно так, как представлено на рисунке слева.

Добавление обработчиков событий

Теперь следует добавить код в наши меню. Мы постараемся не усложнять методы, соответствующие пунктам меню Open (открыть), Save (сохранить) и New (создать новый), и для демонстрациоиных целей будем использовать фиксированный файл. Для того чтобы добавить обработчик событий `Click` в какой-либо пуикт меню, следует просто дважды щелкнуть на нем мышью, как мы это делали для переключателей и других управляющих элементов. Мы добавим обработчики для следующих пунктов меню: New, Open, Save и Exit (выход). Щелчок мышью на разделителе не допускается, кроме того, не должно выполняться никаких действий в том случае, если пользователь щелкает мышью на меню файлов, поэтому для этого пункта меню не предусмотрено никаких обработчиков событий.

Начнем с двойного щелчка мышью на пункте меню New:

```
private void menuItemNew_Click(object sender, System.EventArgs e)
{
    this.rtfText.Clear();
}
```

Параметр sender представляет собой ссылку на пункт меню, на котором произошел щелчок мышью. В данном случае, нам заведомо известно, что таким пунктом является пункт New, однако обработчик события может быть привязан к нескольким пунктам, и тогда этот параметр определяет меню, которое следует использовать.

Повторите процедуру для пунктов Open, Save и Exit:

```
private void menuItemOpen_Click(object sender, System.EventArgs e)
{
   // Загрузка файла
   try
   {
      this.rtfText.LoadFile("../../test.rtf");
   }
   catch (System.Exception err)
   {
      MessageBox.Show("Error while loading:\n" + err.Message);
   }
}
private void menuItemSave_Click(object sender, System.EventArgs e)
{
   // Сохранение файла
   try
   {
      this.rtfText.SaveFile("../../test.rtf");
   }
   catch (System.Exception err)
   {
      MessageBox.Show("Error while saving file:\n" + err.Message);
   }
}
private void menuItemExit_Click(object sender, System.EventArgs e)
{
   // Выход из приложения
   Application.Exit();
}
```

Обратите внимание на то, что мы включили сюда обработку ошибочной ситуации, когда требуемый текстовый файл отсутствует. Существует еще один способ выполнения такой проверки. Пункт меню menuItemFiles содержит и другие пункты нашего основного меню. По этой причине каждый раз, когда приходит запрос на просмотр всех членов данного пункта меню, он отправляет сообщение о событии Popup. Если мы подпишемся на это событие, то окажемся в состоянии осуществлять проверку существования требуемого файла и отключать пункт меню Open в том случае, если таковой не существует. Мы осуществим подписку на это событие, добавив его в конструктор формы:

```
public Form1()
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();

   this.menuItemFiles.Popup += new EventHandler(this.menuItemFiles_Popup);
}
```

Теперь мы создаем собственно обработчик события:

```
private void menuItemFiles_Popup(object sender, System.EventArgs e)
{
   // Позволяет определять, существует ли данный файл, присваивая соответствующее
   // значение свойству Enabled, которое возвращается методом File.Exists.
   this.menuItemOpen.Enabled = System.IO.File.Exists("../../test.rtf");
}
```

Статический метод `Exists` класса `File` возвращает значение логического типа, которое равно `true`, если такой файл существует, и `false` — в противном случае; это именно то, что требуется отобразить в пункте меню Open, поэтому мы просто присваиваем свойству `Enabled` значение, возвращаемое этой функцией.

На этом работа с основным меню завершена, и теперь можно перейти к контекстному меню. Мы добавляем обработчик событий `Click` точно так же, как делали это для основного меню — посредством двойного щелчка мышью на пунктах и внесения следующего кода:

```
private void menuItemBold_Click(object sender, System.EventArgs e)
{
   Font newFont = new Font(rtfText.SelectionFont,
                           (rtfText.SelectionFont.Bold ?
                            rtfText.SelectionFont.Style & ~FontStyle.Bold :
                            rtfText.SelectionFont.Style | FontStyle.Bold));
   rtfText.SelectionFont = newFont;
}

private void menuItemItalic_Click(object sender, System.EventArgs e)
{
   Font newFont = new Font(rtfText.SelectionFont,
                           (rtfText.SelectionFont.Italic ?
                            rtfText.SelectionFont.Style & ~FontStyle.Italic :
                            rtfText.SelectionFont.Style | FontStyle.Italic));
   rtfText.SelectionFont = newFont;
}

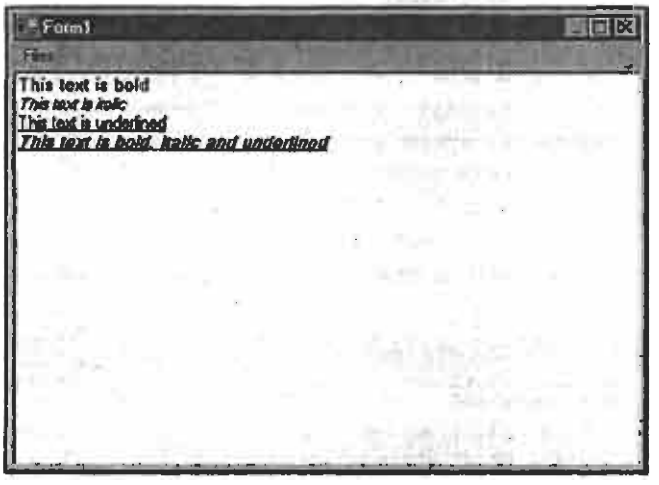
private void menuItemUnderline_Click(object sender, System.EventArgs e)
{
   Font newFont = new Font(rtfText.SelectionFont,
                           (rtfText.SelectionFont.Underline ?
                            rtfText.SelectionFont.Style & ~FontStyle.Underline :
                            rtfText.SelectionFont.Style | FontStyle.Underline));
   rtfText.SelectionFont = newFont;
}
```

В каждой из этих трех функций мы создаем новый шрифт на основе шрифта, который используется в текущий момент в окне RichTextbox. Если стиль шрифта уже задан, то удаляем его из нового шрифта, в противном случае добавляем требуемый стиль в шрифт. После этого присваиваем новый шрифт выбранному тексту.

Единственная строка, предназначенная для создания объектов, выглядит несколько устрашающе, поэтому дадим здесь некоторые пояснения. Если вы чувствуете, что недостаточно знакомы с тринарными операторами, то вам следует обратиться к главе 4. Для пояснений мы возьмем строку из метода `menuItemBold_Click`:

```
1  Font newFont = new Font(rtfText.SelectionFont,
2                          (rtfText.SelectionFont.Bold ?
3                           rtfText.SelectionFont.Style & ~FontStyle.Bold :
4                           rtfText.SelectionFont.Style | FontStyle.Bold));
5  rtfText.SelectionFont = newFont;
```

На строке 2 мы осуществляем проверку, равно ли значение свойства Bold выбранного шрифта true. Если да, то необходимо сформировать новый стиль для создаваемого шрифта, в котором будут сохранены все имеющиеся у него стили за исключением выделения жирным шрифтом, чем и занимается код, расположенный между символами ? и : на строке 3. Код, расположенный на строке 4 — сразу после точки с запятой — будет выполняться в том случае, если текущее значение свойства Bold равно false, причем на этот раз вместо того, чтобы удалять выделение жирным шрифтом, мы добавляем его. Наконец, вся эта последовательность событий заключается в круглые скобки — этот прием весьма часто используется для отделения логики тернарного оператора от остального кода, хотя в данном случае польза от такого выделения не представляется очевидной.



На этом первый пример, посвященный использованию меню, завершен. Если запустить код, то должно получиться нечто вроде представленного на рисунке слева.

Панели инструментов

Хотя меню прекрасно справляются с задачей предоставления доступа ко всему многообразию функциональных возможностей приложения, размещение некоторых пунктов не только в меню, но и на панели инструментов, может дать некоторые выгоды. Это оказывается справедливым для пунктов меню, используемых пользователем наиболее часто, например, Open и Save. Панель ведет к отдельным функциональным возможностям приложения посредством одного щелчка мышью, что, вне всякого сомнения, требует меньших усилий, чем при работе с меню — пользователю намного легче щелкать мышью на постоянно видимой кнопке, чем осуществлять поиск по иерархии различных меню.

Изображение экрана, приведенное ниже, демонстрирует набор линеек инструментов, которые видны автору в процессе написания этой главы в MS Word:



Кнопки, расположенные на панели инструментов, обычно содержат рисунок без текста, хотя и существует возможность использовать кнопки, на которых будет изображено и то, и другое. В качестве панели с кнопками без текста можно привести панели в MS Word (как на рисунке), а в качестве примера панелей с текстом — используемые в MS Internet Explorer. Если вы наведете курсор мыши на какую-либо кнопку и остановите его, то будет выведено некоторое пояснение относительно предназначения данной кнопки, особенно в тех случаях, когда на ней имеется только изображение и отсутствует какой-либо текст.

В отличие от управляющих элементов типа "меню", управляющий элемент `Toolbar` не является просто контейнером для других объектов. Можно задать непосредственно некоторые его свойства, например, координаты его расположения на экране. Все кнопки на панели инструментов являются полноправными объектами. Каждая кнопка элемента `Toolbar` представляет собой объект `ToolbarButton`, в котором содержится информация о том, какой должен выводиться рисунок, каков стиль текста, который выводится/не выводится вместе с рисунком, а также некоторые другие пояснения.

Сначала опишем свойства и события управляющего элемента `Toolbar`, а затем перейдем к обсуждению `ToolbarButton`.

Свойства управляющего элемента Toolbar

Свойства элемента `Toolbar` позволяют управлять тем, как и где этот элемент будет выводиться на экран. Они также регулируют некоторые установки, касающиеся вывода кнопок управляющего элемента на экран; такие установки являются едиными для всех кнопок, содержащихся на данной панели:

Имя	Режим доступа	Описание
`Appearance`	Чтение/запись	Управляет внешним видом всех кнопок, содержащихся в данном элементе. Оно может принимать значения либо `Flat` (плоский), либо `Normal` (нормальный). Установка, имеющая значение `Normal` означает, что вокруг кнопки будет нарисована объемная граница.
`AutoSize`	Чтение/запись	Присваивание этому свойству значения `false` позволяет изменять размеры управляющего элемента. При значении по умолчанию `true` элемент самостоятельно определяет собственные размеры с тем, чтобы было достаточно места для вывода рисунков и текста.
`Buttons`	Только чтение	Возвращает семейство кнопок, содержащихся в данном элементе.
`ButtonSize`	Чтение/запись	Позволяет изменять размеры самих кнопок. Если свойство `AutoSize` имеет значение, равное `true`, то данное свойство игнорируется.
`Divider`	Чтение/запись	Если этому свойству присвоено значение `true`, то будет выведена верхняя граница управляющего элемента `Toolbar`.
`DropDownArrows`	Чтение/запись	Если в управляющем элементе имеются кнопки, которые представляют раскрывающиеся меню, то это свойство позволяет определить, будет ли стрелка, предназначенная для раскрытия этого меню, рисоваться с правой стороны.
`ImageList`	Чтение/запись	Список рисунков, в котором хранятся рисунки, используемые кнопками данного управляющего элемента.
`ShowToolTips`	Чтение/запись	Присваивание этому свойству значения `true` приведет к выводу пояснений для всех кнопок, содержащихся в данном управляющем элементе.
`Wrappable`	Чтение/запись	Если это свойство имеет значение `true`, а длина панели инструментов слишком велика для того, чтобы все входящие в нее кнопки могли быть выведены на одной строке, то она будет перенесена на следующую строку.

События управляющего элемента Toolbar

В классе Toolbar существуют два события, не являющиеся производными от класса Control. Они генерируются, когда происходит щелчок мышью на кнопке или на стрелке, предназначенной для раскрытия меню. Единственный обработчик событий должен справляться с нажатиями всех возможных кнопок на панели, поэтому он должен быть способным различать нажатие одной кнопки от нажатия другой:

Имя	Описание
ButtonClick	Происходит при нажатии любой кнопки, входящей в состав панели инструментов. Для того чтобы определить, какая именно кнопка была нажата, следует воспользоваться свойством Button параметра ToolbarButtonClickEventArgs, который передается обработчику событий. С помощью этого свойства можно определять действия, которые необходимо предпринять.
ButtonDropDown	Наступает, если кнопке панели инструментов вместе со ее свойством Style присваивается значение DropDownButton или если происходит щелчок мышью на соответствующей стрелке. В этом случае также возможно использовать ToolbarButtonClickEventArgs для определения того, какая именно кнопка нажата и, следовательно, какие действия требуется предпринять.

Свойства класса ToolbarButton

Класс ToolbarButton определяет отдельные установки для каждой кнопки панели инструментов, включая рисунок, стиль, текст и пояснение по использованию данного инструмента. Интересная особенность ToolbarButton заключается в том, что если стиль имеет значение DropDownButton, то этот класс может содержать меню (MenuItem), в котором содержатся пункты, подлежащие выводу:

Имя	Режим доступа	Описание
DropDownMenu	Чтение/запись	Дает возможность присваивать MenuItem данной кнопке. Если свойство Style этой кнопки имеет значение DropDownButton (см. ниже), то при нажатии данной кнопки будет выводиться MenuItem.
Enabled	Чтение/запись	Установив значение этого свойства false, можно сделать данную кнопку недоступной для пользователя. Не забудьте, что в этом случае при использовании для кнопки растровых изображений рисунки можно выводить только в монохромном изображении.
ImageIndex	Чтение/запись	Индекс рисунка, который следует использовать. Список рисунков привязывается к управляющему элементу Toolbar.
PartialPush	Чтение/запись	Может использоваться, если стиль данной кнопки определен как ToggleButton. Если присвоить этому свойству значение true, то кнопка будет выводиться серым цветом. В отличие от случая, когда свойству Enabled присваивается значение false, эта установка приводит к тому, что вся поверхность кнопки становится нечеткой.
Pushed	Чтение/запись	Если стиль ToolbarButton имеет значение ToggleButton, то присвоив этому свойству значение true, можно добиться того, что кнопка будет выводиться нажатой.

Продолжение таблицы

Имя	Режим доступа	Описание
Style	Чтение/запись	Можно использовать четыре различных стиля: • PushButton: Кнопка выводится в нормальном виде. • ToggleButton: Кнопка данного стиля может быть нажата и будет оставаться в таком положении до тех пор, пока не будет нажата еще раз. Пример такой кнопки — кнопка Bold, использующаяся в панелях инструментов MS Word. • Separator: Кнопка данного стиля вообще не выводится; она создает небольшое пространство между кнопками. • DropDownButton: Кнопке с таким стилем может быть присвоено MenuItem. Если в нем содержатся подпункты, они будут использоваться для создания меню, которое будет раскрываться при нажатии на данную кнопку.
Text	Чтение/запись	Текст, который должен выводиться на кнопке, находящейся на панели инструментов.
ToolTipText	Чтение/запись	Текст пояснения для данной кнопки.

Практикум: пример использования панели инструментов

Мы расширим тот пример, который использовался при обсуждении меню, за счет включения в него панели инструментов. Эта простая панель инструментов будет содержать три кнопки, каждая из которых соответствует одному пункту контекстного меню — выделение жирным шрифтом, курсивом и подчеркиванием. Мы включим туда четвертую кнопку с раскрывающимся меню, с помощью которой можно выбирать шрифт.

> *Автору не удалось найти растровый файл для кнопки Font среди графических файлов, инсталлированных в Visual Studio.NET. Вы можете использовать для этой цели произвольный графический файл, либо скачать текст этой книги, который находится на www.wrox.com и в котором можно найти файл, использованный автором.*

Начнем с того, что внесем следующие изменения в код MenuExample, который мы создали в начале главы:

1. Поскольку мы собираемся добавлять панель инструментов, необходимо изменить свойство Dock управляющего элемента RichTextbox на None, а вместо него присвоить свойству Anchor значение Left, Top, Right, Bottom.

2. Добавьте управляющие элементы ImageList и еще одно MainMenu в форму с помощью двойного щелчка мышью на соответствующих элементах в окошке инструментов. Назовите их ImageListToolbar и MainMenuFonts соответственно. В элементе ImageList будут храниться рисунки, которые мы будем использовать для кнопок, а в MainMenu — меню, которое будет использовано в качестве раскрывающегося списка.

Имя	Текст
menuItemFonts	
menuItemMS	MS Sans Serif
menuItemTimes	Times New Roman

3. Добавьте три пункта MenuItem в управляющий элемент MainMenuFonts и присвойте свойствам значения, согласно таблице слева.

4. Добавьте в ImageList четыре рисунка, щелкнув мышью на кнопке, расположенной справа от свойства Images в управляющем элементе ImageList. Три из этих рисунков могут быть найдены в поддиректории Common7 той директории, в которой вы инсталлировали Visual Studio:

   <Директория, содержащая VS.NET>\Common7\Graphics\bitmaps\Tlbr_W95 под именами BLD.BMP, ITL.BMP и UNDRLN.BMP.

   Для последнего пункта меню можно использовать произвольный рисунок, например, содержащийся в исходной программе на сайте издательства Wrox и используемый в данной книге (Font.bmp).

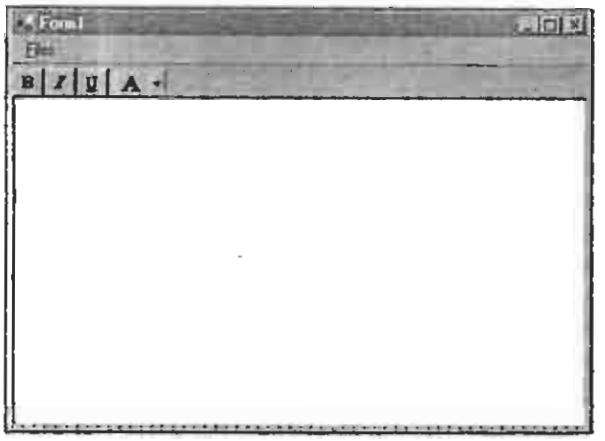
5. Щелкните два раза мышью на управляющем элементе Toolbar в окне инструментов и добавьте ее к форме. Назовите его toolbarFonts. Эта панель инструментов может частично заслонить RichTextbox. Для исправления данной ситуации щелкните мышью на RichTextbox и переместите его ниже панели инструментов.

6. Присвойте свойству ImageList панели инструментов значение ImageListToolbar.

7. Выберите свойство Button управляющего элемента Toolbar и щелкните на кнопке, расположенной справа от него, что позволит добавить кнопки к элементу Toolbar.

8. Добавьте к панели инструментов пять кнопок и очистите свойство Text у каждой из них. Присвойте свойствам этих пяти кнопок значения в соответствии с приведенной ниже таблицей:

ToolbarButton	Свойства
Первая кнопка	Name = toolBarButtonBold
	ImageIndex = индекс рисунка, обозначающего жирный шрифт (B)
	Style = ToggleButton
Вторая кнопка	Name = toolBarButtonItalic
	ImageIndex = индекс рисунка, обозначающего курсив (I)
	Style = ToggleButton
Третья кнопка	Name = toolBarButtonUnderline
	ImageIndex = индекс рисунка, обозначающего подчеркивание (U)
	Style = ToggleButton
Четвертая кнопка	Name = toolBarButtonSeparator
	Style = Separator
Пятая кнопка	Name = toolBarButtonFonts
	ImageIndex = индекс последнего оставшегося рисуика
	Style = DropDownButton
	DropDownMenu = menuItemFonts



Теперь форма должна выглядеть приблизительно так, как показано на рисунке слева.

Заметьте, что этот проект является частью кода для настоящей книги, его можно загрузить в виде проекта Visual Studio.NET с именем `ToolBars`.

Добавление обработчиков событий

Теперь можно добавить обработчик события `ButtonClick` элемента `Toolbar`, для чего достаточно дважды щелкнуть мышью на управляющем элементе:

```
private void toolBarFonts_ButtonClick(object sender,
                                      System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
   Font newFont;

   // Переключение на индекс соответствующей кнопки в семействе Buttons
   // управляющего элемента Toolbar
   switch (toolBarFonts.Buttons.IndexOf(e.Button))
   {
      case 0:                          // Жирный шрифт
         if (e.Button.Pushed)
            // Создание нового шрифта с выделением жирным шрифтом
            newFont = new Font(rtfText.SelectionFont,
                               rtfText.SelectionFont.Style | FontStyle.Bold);
         else
            // Создание нового шрифта без выделения жирным шрифтом
            newFont = new Font(rtfText.SelectionFont,
                               rtfText.SelectionFont.Style & ~FontStyle.Bold);
         rtfText.SelectionFont = newFont;
         break;
      case 1:                          // Курсив
         if (e.Button.Pushed)
            // Создание нового шрифта с выделением курсивом
            newFont = new Font(rtfText.SelectionFont,
                               rtfText.SelectionFont.Style | FontStyle.Italic);
         else
            // Создание нового шрифта без выделения курсивом
            newFont = new Font(rtfText.SelectionFont,
                               rtfText.SelectionFont.Style & ~FontStyle.Italic);
            rtfText.SelectionFont = newFont;
            break;
      case 2:                          // Подчеркивание
         if (e.Button.Pushed)
            // Создание нового шрифта с подчеркиванием
            newFont = new Font(rtfText.SelectionFont,
                               rtfText.SelectionFont.Style | FontStyle.Underline);
         else
            // Создание нового шрифта без подчеркивания
            newFont = new Font(rtfText.SelectionFont,
                               rtfText.SelectionFont.Style & ~FontStyle.Underline);
         rtfText.SelectionFont = newFont;
         break;
   }
}
```

В этой функции мы сначала определяем индекс той кнопки, которая была нажата. Это необходимо, поскольку все кнопки семейства `Buttons` элемента `Toolbar` используют данное событие для оповещения о нажатии. После того, как та кнопка, которая была нажата, определена, проверяем ее свойство `Pushed`, для того чтобы понять, следует ли установить или удалить выбранный стиль шрифта, и в соответствии с этой информацией создаем новый стиль. В завершение присваиваем новый шрифт свойству `SelectionFont` управляющего элемента `rtfText`.

Последний момент, касающийся тех экземпляров класса `ToolbarButton`, которые управляют стилем шрифта, заключается в том, что они должны правильно отражать шрифт, используемый для текущего выделения. С этой целью в управляющий элемент `RichTextbox` следует добавить событие `SelectionChanged`. Сначала мы подписываемся на это событие в конструкторе:

```
public Form1()
{
   InitializeComponent();

   this.menuItemFiles.Popup += new EventHandler(this.menuItemFiles_Popup);
   this.rtfText.SelectionChanged +=
                        new EventHandler(this.rtfText_SelectionChanged);
}
```

Затем мы добавляем собственно обработчик событий:

```
private void rtfText_SelectionChanged(object sender, System.EventArgs e)
{
   // Корректная установка кнопок на панели инструментов в нажатом
   // или ненажатом состоянии
   this.toolBarButtonBold.Pushed = rtfText.SelectionFont.Bold;
   this.toolBarButtonItalic.Pushed = rtfText.SelectionFont.Italic;
   this.toolBarButtonUnderline.Pushed = rtfText.SelectionFont.Underline;
}
```

Элемент `RichTextbox` позволяет получать информацию о стиле текущего шрифта, поэтому мы просто присвоим соответствующие значения свойству `Enabled` для кнопок на панели инструментов.

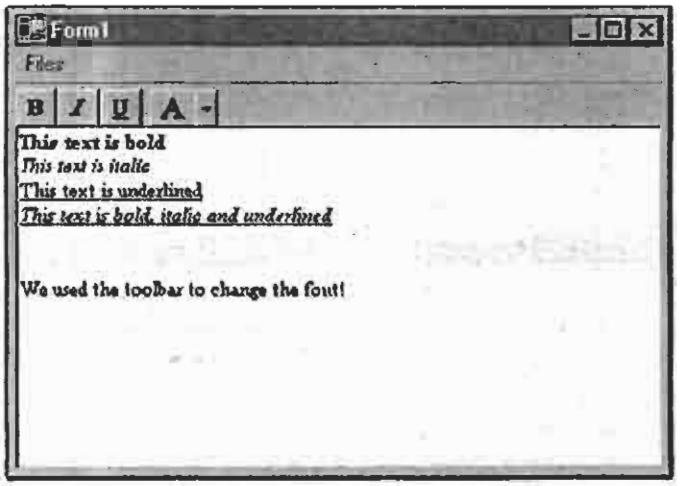
Все, что теперь осталось сделать,— это обработать событие "щелчок мышью" для двух пунктов меню в кнопке типа `DropDownButton`. Для этого добавим события `Click` в пункты меню — точно так же, как делали в примере использования меню, приведенном выше в этой главе (двойным щелчком мыши на соответствующем пункте):

```
private void menuItemMS_Click(object sender, System.EventArgs e)
{
   // Создание нового шрифта на основе требуемого семейства шрифтов.
   Font newFont = new Font("MS Sans Serif", rtfText.SelectionFont.Size,
                           rtfText.SelectionFont.Style);
   rtfText.SelectionFont = newFont;
}

private void menuItemTimes_Click(object sender, System.EventArgs e)
{
   // Создание нового шрифта на основе требуемого семейства шрифтов.
   Font newFont = new Font("Times New Roman", rtfText.SelectionFont.Size,
                           rtfText.SelectionFont.Style);
   rtfText.SelectionFont = newFont;
}
```

Мы создаем новый шрифт из того семейства шрифтов, который мы хотим использовать, а затем присваиваем свойство `SelectionFont` вновь созданному шрифту.

Если вы используете версию Visual Studio.NET Beta 2, то при попытке откомпилировать данный пример, вы увидите несколько предупреждений. Это не является следствием каких-либо действий с вашей стороны, скорее, это следствие действий со стороны компании Microsoft. Дело в том, что код, генерируемый для экземпляров `ToolbarButton`*, включает в себя строку, в которой происходит присваивание свойства* `ItemData`*. Компания Microsoft пометила это свойство как устаревшее, которое должно быть удалено когда-нибудь в будущем. В предупреждениях содержится совет использовать вместо него свойство* `Tag`*. Не стоит беспокоиться и удалять соответствующие строки — редактор все равно вернет их на место, как только вы откроете окно разработки.*

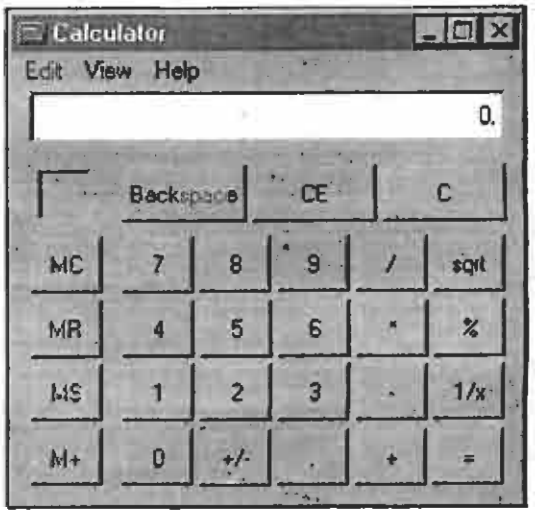


Теперь откомпилируйте и запустите приложение (см. рис. слева), а затем попытайтесь выполнить в нем какие-либо действия.

SDI- и MDI-приложения

Традиционно существует три разновидности приложений, которые могут создаваться для MS Word, а именно:

- Приложения, основанные на диалоговом окне. Такие приложения предоставляются пользователям в виде единого диалогового окна, с помощью которого может быть осуществлен доступ ко всем функциональным возможностям.
- Однодокументные интерфейсы (Single Document Interface, SDI). Такие приложения предоставляются пользователям в виде меню, одной или нескольких линеек инструментов и одного окна, в котором пользователь может выполнять определенные действия.
- Многодокументные интерфейсы (Multi-Document Interface, MDI). Такие приложения представляются пользователям в таком же виде, что SDI-приложения, однако обладают способностью одновременно поддерживать несколько открытых окон.



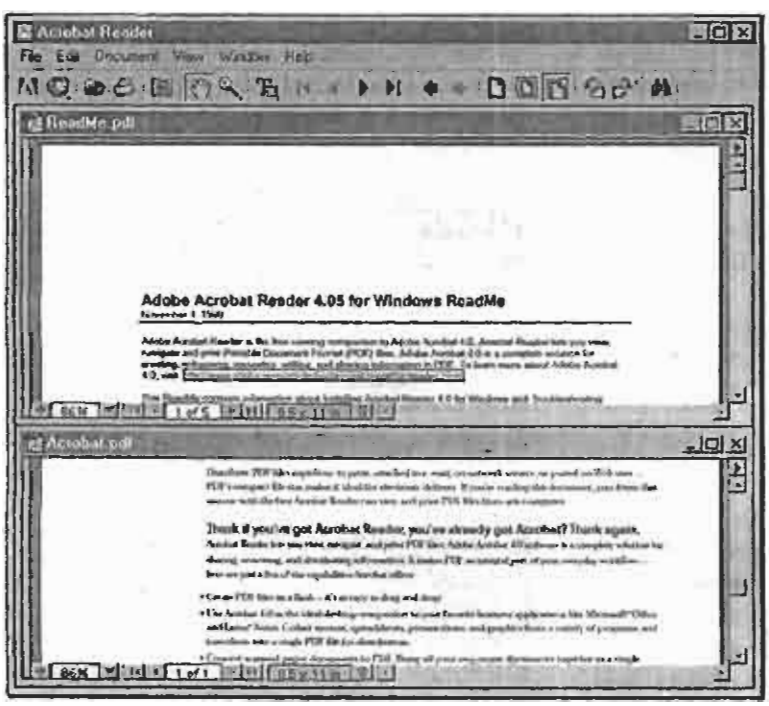
Приложения, основанные на диалоговом окне, обычно представляют собой небольшие одноцелевые приложения, которые ориентированы либо для решения конкретной задачи, требующей ввода небольшого количества данных, либо для работы с какими-то необычными типами данных. В качестве примера такого приложения можно привести Calculator (калькулятор), поставляемый вместе с MS Windows (см. рис. справа).

Однодокументные интерфейсы, как правило, предназначаются для решения какой-то одной конкретной задачи, при этом они позволяют пользователю загружать в приложение единственный документ, с которым он и будет вести работу. Эта задача предполагает выполнение пользователем большого количества действий, и зачастую пользователю могут потребоваться возможности, позволяющие сохранять или загружать плоды своего труда. Хорошим примером SDI-приложений могут служить MS Paint и WordPad (см. рис. справа), также поставляемые совместно с MS Windows.



Однако такие приложения допускают открытие только одного документа в каждый конкретный момент времени, поэтому если пользователю требуется открыть второй документ, то ему будет необходимо открывать новый экземпляр SDI-приложения, у которого будет отсутствовать связи с первым документом и, следовательно, конфигурация, созданная для первого экземпляра, не окажет никакого влияния на конфигурацию второго. Например, вы в MS Paint выбрали красный цвет в качестве цвета рисования, затем открываете второй экземпляр MS Paint, а здесь в качестве цвета, используемого для рисования, выбирается цвет по умолчанию. Он будет черный.

Многодокументные интерфейсы почти полностью аналогичны SDI-приложениям за исключением того, что они обладают возможностью поддерживать более одного открытого документа в различных окнах, которые могут быть открыты одновременно. Одним из простых признаков MDI-приложения является наличие пункта Window на правой стороне линейки меню перед пунктом Help. Примерами MDI-приложений служат Adobe Acrobat Reader (см. рис. слева) и MS Word 97.



Четвертый тип приложений представлен MS Office 2000. Этот тип является смесью SDI- и MDI-приложений: окна, предоставляемые пользователю, имеют различное местоположение, и каждое окно отображается в линейке заданий. Такое приложение представляет собой несколько MDI-приложений, поскольку основное приложение не будет закрыто до тех пор, пока не будут закрыты все окна, а с помощью пункта меню Windows можно выбирать, какой именно из открытых документов будет просматриваться, хотя при этом собственно пользовательский интерфейс представляет собой SDI-интерфейс.

В этой главе мы сосредоточимся на создании MDI-приложений и тех задачах, которые при этом возникают. Дело в том, что любое SDI-приложение в основном представляет собой подмножество MDI-приложения, поэтому, если вы умеете создавать MDI-приложения, вы также сможете создавать и SDI-приложения. В главе 15 мы создадим простое SDI-приложение, которое будет использоваться для демонстрации Windows Common Dialogs (общие диалоговые окна Windows).

Создание MDI-приложений

Для создания MDI, во-первых, необходимо, чтобы решаемая пользователем задача требовала одновременно несколько открытых документов. Примером задач такого рода является текстовый редактор или, как в приведенном выше примере, программа просмотра документов. Во-вторых, необходимо предусмотреть панели инструментов для наиболее часто выполняемых в приложении операций, таких как изменение стиля шрифта, загрузка и сохранение документов. В-третьих, обязательно следует предусмотреть пункт меню Window, который позволял бы пользователю изменять положение открытых окон друг относительно друга (налагая их друг на друга в виде черепицы или каскада) и предоставлял бы ему список всех открытых окон. Еще одной особенностью MDI-приложений является то, что если имеется некоторое открытое окно и в этом окне существует некоторое меню, то оно должно быть интегрировано в основное меню приложения.

Любое MDI-приложение состоит по крайней мере из двух разных окон. Первое окно называется **MDI-контейнером**, а окно, которое может быть открыто в контейнере, называется **дочерним MDI-окном**. При упоминании первого окна будут использоваться взаимозаменяемые термины "MDI-контейнер" и "основное окно", а при упоминании второго — термины "дочернее MDI-окно" или просто "дочернее окно".

При создании MDI-приложения следует начинать с того же, с чего начинается создание любого другого приложения,— с создания Windows Application в Visual Studio. Чтобы превратить основное окно приложения из формы в MDI-контейнер, достаточно просто присвоить свойству формы `IsMdiContainer` значение `true`. При этом произойдет изменение цвета фона, указывающее на то, там не следует размещать видимые управляющие элементы, хотя такая возможность по-прежнему существует и может при определенных обстоятельствах оказаться полезной.

Для создания дочернего окна следует добавить в проект новую форму, выбрав Windows Form из диалогового окна, которое открывается при выборе пункта меню Project | Add New Item. Эта форма становится дочерним окном, когда его свойству `MdiParent` присваивается ссылка на основное окно. Этому свойству нельзя присваивать значение с помощью панели Properties, это необходимо выполнять программным путем.

До того, как появится возможность выводить MDI-приложение на экран в его основном виде, нужно выполнить еще две вещи. Необходимо передать MDI-контейнеру информацию о том, какие окна должны выводиться, а затем вывести их, для чего следует просто создать новый экземпляр формы, которую вы собираетесь выводить, а затем вызвать для нее метод `Show()`. Конструктор формы, предназначенной для вывода в качестве дочернего окна, должен привязаться к родительскому контейнеру. Это достигается за счет присваивания его свойства `MdiParent` экземпляру MDI-контейнера.

Прежде чем переходить к выполнению более сложных задач, разберем небольшой пример, в котором выполняются все эти шаги.

Практикум: создание MDI-приложения

1. Создайте новое Windows Application и назовите его `MdiBasic`.
2. Выберите форму и задайте свойства, сведенные в таблицу справа.
3. Добавьте новую форму к решению, выбрав пункт Windows Form из меню Project | Add New Item. Назовите эту форму `frmChild`.

Свойство	Значение
Name	frmContainer
IsMdiContainer	True
Text	MDI Basic
WindowState	Maximized

Весь код, необходимый для того, чтобы вывести дочернюю форму, располагается в конструкторах форм. Сначала мы займемся конструктором дочернего окна:

```
public frmChild(MdiBasic.frmContainer parent)
{
    InitializeComponent();

    // Присваивание контейнеру родителя данной формы
    this.MdiParent = parent;
}
```

Чтобы привязать дочернюю форму к MDI-контейнеру, ее необходимо зарегистрировать в контейнере. Это достигается присваиванием свойству `MdiParent` данной формы соответствующего значения, как показано в приведенном выше коде. Заметьте, что используемый конструктор включает в себя параметр `parent`.

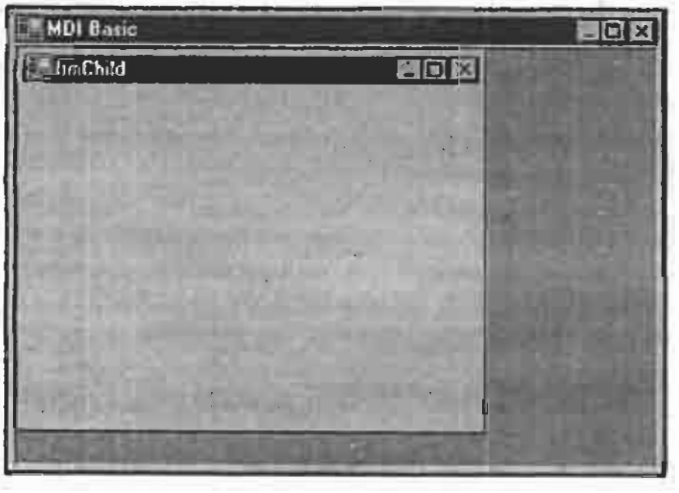
Поскольку в C# не предусмотрены конструкторы по умолчанию для класса, в котором описывается его собственный конструктор, приведенный выше код позволяет предотвратить создание экземпляра данной формы, не привязанного к MDI-контейнеру.

Выполним вывод формы на экран в конструкторе MDI-контейнера:

```
public frmContainer()
{
    InitializeComponent();

    // Создание нового экземпляра дочерней формы
    MdiBasic.frmChild child = new MdiBasic.frmChild(this);

    // Вывод созданной формы
    child.Show();
}
```



Мы создаем новый экземпляр дочернего класса и передаем его конструктору посредством ключевого слова `this`, которое представляет собой текущий экземпляр класса MDI-контейнера. Затем вызываем метод `Show()` для вывода нового экземпляра дочерней формы. Если нужно вывести более одного дочернего окна, то потребуется повторить две выделенные выше в коде строки для каждого окна.

Запустите код, и вы увидите приблизительно то, что показано на рисунке слева.

Это не самый впечатляющий пользовательский интерфейс, однако он вполне

подходит в качестве первоосновы. Следующий пример представляет несложный текстовый редактор, использующий меню и панели инструментов.

Практикум: создание текстового редактора с использованием MDI

Давайте сначала создадим основной проект, а затем обсудим происходящее:

Свойства	Значение
Name	frmContainer
IsMdiContainer	True
Text	Simple Text Editor
WindowState	Maximized

1. Создайте новый проект Windows Application и назовите его `SimpleTextEditor`. Присвойте свойствам формы значения, сведенные в таблицу слева.
2. Переименуйте файл, в котором содержится форма (именно файл, а не свойство) на `frmContainer.cs` и внесите следующие изменения в метод `Main()`:

```
static void Main()
{
    Application.Run(new frmContainer());
}
```

3. Вставьте существующий объект в проект (Project | Add Existing Item). Найдите образец панели инструмеитов и вставьте форму (ее можно назвать `Form1.cs`).
4. Назовите вновь вставленную форму `frmEditor`. Измените имя файла, в котором она находится, на `frmEditor.cs`. Присвойте свойству `Text` значение `Editor`.

   Эти шаги очень похожи на шаги, предпринимавшиеся при создании приложения MDI Basic; отличие заключается только лишь в использовании уже существующей формы вместо создания новой. Однако самое интересное следует далее.
5. Перейдите к коду формы, в которой содержится редактор. Измените имя пространства имен на `SimpleTextEditor` и замените все экземпляры `Form1` на `frmEditor`.
6. Найдите метод `Main()` формы и удалите его.
7. Измените конструктор формы, в которой содержится редактор, следующим образом:

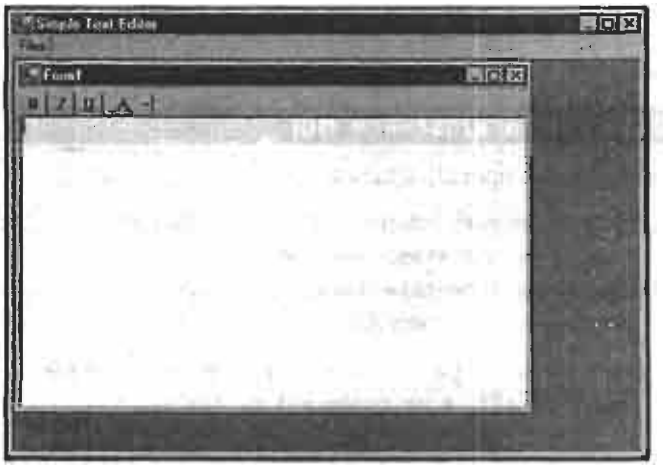
```
public frmEditor(SimpleTextEditor.frmContainer parent)
{
    InitializeComponent();

    // Привязка к родителю
    this.MdiParent = parent;
}
```

8. Измените конструктор формы `frmContainer` следующим образом:

```
public frmContainer()
{
    InitializeComponent();

    SimpleTextEditor.frmEditor newForm = new frmEditor(this);
    newForm.Show();
}
```



При запуске данного приложения получается что-то похожее на изображенное на рисунке слева.

Произошло нечто необъяснимое. Мы не создавали никаких меню для контейнера, и тем не менее меню Files находится справа вверху. Почему так получилось? Ответ заключается в том, что было создано меню для формы, содержащей редактор, а так как эта форма является в настоящий момент дочерней формой контейнера, то пункты основного меню оказались объединенными. Поскольку в контейнере пока отсутствуют свои собственные меню, то в этом меню всего один пункт.

Меню, содержащиеся в дочерних окнах, являются специфическими для данного окна. В отличие от них меню Files должно быть общим для всех окон — оно не должно содержаться в дочернем окне. Причина этого условия станет очевидной, если вы закроете окно редактора,— меню Files исчезнет! Чтобы изменить такое поведение, следует предпринять несколько дополнительных шагов:

9. Удалите меню с именем `MainMenuFiles` из формы, содержащей редактор, щелкнув на нем правой кнопкой мыши и выбрав Delete. В появившемся окне с сообщением нажмите Yes, чтобы его закрыть.

10. Перейдите к коду и удалите следующие обработчики событий для данного меню:
    - `menuItemFiles_Popup`
    - `menuItemSave_Click`
    - `menuItemNew_Click`
    - `menuItemOpen_Click`
    - `menuItemExit_Click`

11. Удалите из конструктора строку, в которой происходит подписка на событие `menuItemFiles_Popup`.

    Более подходящим для формы, содержащей редактор, будет меню Edit. Для его создания выполните следующие шаги:

12. Добавьте новое меню в форму, содержащую редактор. Назовите этот управляющий элемент `MainMenuEdit`.

13. Добавьте в него семь пунктов и присвойте значения свойствам:

Имя	Текст	Порядок объединения
`menuItemEdit`	`&Edit`	`0`
`menuItemUndo`	`&Undo`	Не задан
`menuItemRedo`	`&Redo`	Не задан
`menuItemSeparatorEdit0`	`-`	Не задан
`menuItemCut`	`Cu&t`	Не задан
`menuItemCopy`	`&Copy`	Не задан
`menuItemPaste`	`&Paste`	Не задан

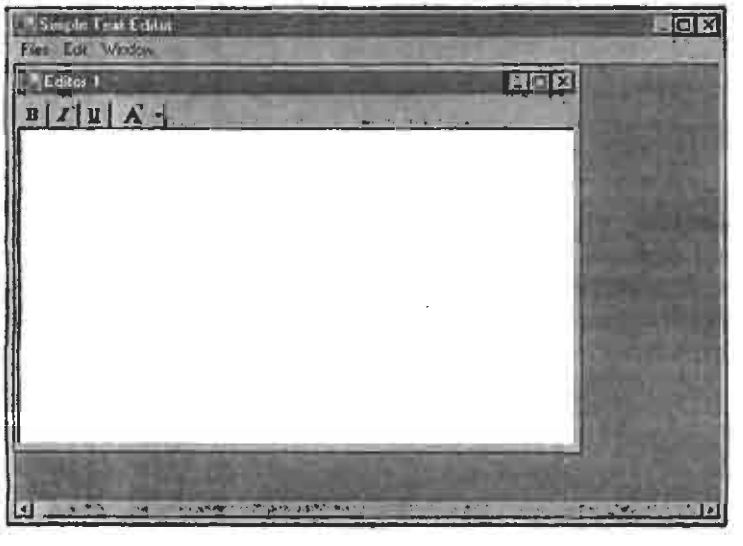
14. Присвойте свойству `Menu` формы, содержащей редактор, значение `MainMenuEdit`. Это меню будет объединено с меню основного окна.

Теперь можно приступить к созданию меню, которое будет выводиться на экран независимо от того, какое именно окно выбрано в настоящий момент. Для того чтобы добавить меню в контейнер, необходимо выполнить следующие шаги:

15. Добавьте к форме `frmContainer` меню `MainMenu`. Назовите его `MainMenuContainer`.

16. Добавьте в это меню указанные ниже пункты. Обратите внимание на то, что пункт `MenuItemWindow` должен быть пунктом меню самого верхнего уровня — то есть он должен располагаться справа от меню Files:

Имя	Текст	Свойство MdiList	Самый верхний уровень	Порядок объединения
menuItemFiles	&Files	False	Да	0
menuItemNew	&New	False	Нет	Не задан
menuItemOpen	&Open	False	Нет	Не задан
menuItemClose	&Close	False	Нет	Не задан
menuItemSepFiles0	-	False	Нет	Не задан
menuItemSave	&Save	False	Нет	Не задан
menuItemSaveAll	Save &All	False	Нет	Не задан
menuItemSepFiles1	-	False	Нет	Не задан
menuItemExit	E&xit	False	Нет	Не задан
menuItemWindow	&Window	False	Да	1
menuItemTile	&Tile	False	Нет	Не задан
menuItemCascade	&Cascade	False	Нет	Не задан
menuItemSepWindow0	-	False	Нет	Не задан
menuItemWindowsOpen	Open Windows	True	Нет	Не задан



Если запустить приложение, то можно увидеть, что меню Edit помещено между меню Files и Window. Причина произошедшего заключается в том, что мы задали 1 в качестве значения свойства `MergeOrder` управляющего элемента `MenuItemWindow`, а в качестве же значения свойства `MergeOrder` меню Edit — 0. Поскольку свойство `MergeType` имеет значение `Add`, то меню Edit добавляется справа от всех пунктов меню самого верхнего уровня, у которых значение свойства `MergeOrder` меньше значения свойства `MergeOrder` меню Edit или равно ему (см. рис. слева).

Еще один момент, достойный внимания, заключается в том, что пункт Open меню Windows содержит подпункт с окошком выбора. Текст этого подпункта совпадает с текстом формы, содержащей редактор. В названном меню будут перечислены все открытые окна — для это не потребуется написать ни единой строки кода. Такое положение является результатом присвоения свойству `MdiList` пункта меню `menuItemWindowsOpen` значения `true` (см. рис. справа).



Добавление обработчиков событий

Для написания кода сначала нужно создать обработчик меню для меню Edit, щелкнув два раза мышью на всех пунктах этого меню за исключением пункта самого верхнего уровня и разделителя и введя следующий код:

```
private void menuItemUndo_Click(object sender, System.EventArgs e)
{
   rtfText.Undo();
}

private void menuItemRedo_Click(object sender, System.EventArgs e)
{
   rtfText.Redo();
}

private void menuItemCut_Click(object sender, System.EventArgs e)
{
   rtfText.Cut();
}

private void menuItemCopy_Click(object sender, System.EventArgs e)
{
   rtfText.Copy();
}

private void menuItemPaste_Click(object sender, System.EventArgs e)
{
   rtfText.Paste();
}
```

В `RichTextbox` есть методы, точно соответствующие всем пунктам меню, поэтому просто вызываем эти функции. Как и в примере с `MenuExample`, рассматривавшемся ранее, необходимо проверить, должно ли данное меню быть отключенным. Эта проверка осуществляется в обработчике событий `Popup`. Сначала мы вносим в конструктор формы `frmEditor`:

```
public frmEditor(SimpleTextEditor.frmContainer parent)
{
   InitializeComponent();

   // Привязка к родителю
   this.MdiParent = parent;
```

```
            // Подписка на событие Popup меню Edit
            this.menuItemEdit.Popup += new EventHandler(this.menuItemEdit_Popup);
        }
```

Затем добавляем собственно обработчик событий:

```
private void menuItemEdit_Popup(object sender, System.EventArgs e)
{
    // Если выбранный текст отсутствует, то его нельзя вырезать.
    this.menuItemCut.Enabled = rtfText.SelectedText.Length > 0 ? true : false;

    // Если выбранный текст отсутствует, его нельзя копировать.
    this.menuItemCopy.Enabled = rtfText.SelectedText.Length > 0 ? true : false;

    // Метод CanPaste управляющего элемента RichTextbox сообщает о том,
    // имеется ли какая-либо информация, предназначенная для вставки
    this.menuItemPaste.Enabled =
                    rtfText.CanPaste(DataFormats.GetFormat(DataFormats.Rtf));

    // Свойство CanUndo управляющего элемента RichTextbox сообщает о том,
    // имеется ли возможность отмены какого-либо действия
    this.menuItemUndo.Enabled = rtfText.CanUndo;

    // Свойство CanRedo управляющего элемента RichTextbox сообщает о том,
    // имеется ли возможность выполнения какого-либо отмененного действия
    this.menuItemRedo.Enabled = rtfText.CanRedo;
}
```

Управляющий элемент `RichTextbox` оказывает исчерпывающую помощь. Можно проверить все пункты меню, чтобы выяснить, должны они быть отключены или нет, простым вызовом функции или просмотром значения свойства. Единственное, что требует пояснений,— это `CanPaste()`. Этот метод получает текст, который вы собираетесь вставлять, и возвращает логическое значение, которое равно `true` в том случае, если такая вставка оказывается возможной.

Теперь обратимся к меню в MDI-контейнере и начнем с пункта New меню Files. Когда происходит щелчок мышью на этом пункте, требуется создание нового окна. Подобная процедура выполняется в конструкторе. Но при этом возникает проблема. На данный момент все окна обладают одним и тем же заголовком — Editor, что не позволяет различать их в списке `MdiList`. Для того чтобы исправить ситуацию, добавим в конструктор формы `Editor` еще один параметр, в котором будем передавать текст, предназначенный для вывода в качестве заголовка вновь создаваемого окна:

```
public frmEditor(SimpleTextEditor.frmContainer parent, string caption)
{
    InitializeComponent();

    // Привязка к родителю
    this.MdiParent = parent;

    // Задание заголовка
    this.Text = caption;

    // Подписка на событие Popup меню Edit
    this.menuItemEdit.Popup += new EventHandler(this.menuItemEdit_Popup);
    this.rtfText.SelectionChanged += new
                                EventHandler(this.rtfText_SelectionChanged);
}
```

Это изменение означает, что необходимо изменить вызов конструктора, который создается в конструкторе контейнера:

```
public frmContainer()
{
   InitializeComponent();

   SimpleTextEditor.frmEditor newForm = new frmEditor(this, "Editor 1");
   newForm.Show();
}
```

Теперь можно создать новые окна. Щелкните два раза мышью на пункте меню New и добавьте следующий код:

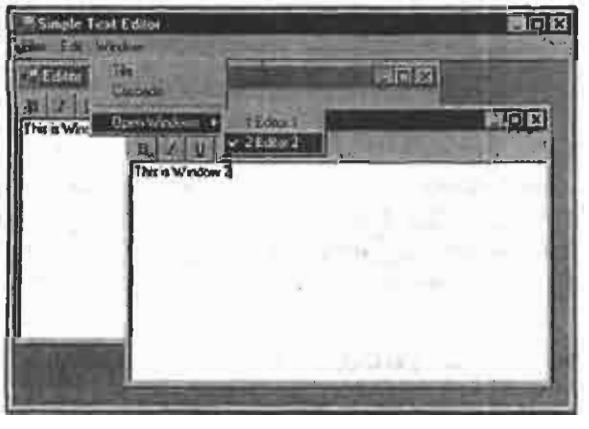
```
private void menuItemNew_Click (object sender, System.EventArgs e)
{
   string caption = "Editor " + nextFormNumber++;          // Заголовок

   // Создание нового окна
   SimpleTextEditor.frmEditor newForm = new
                              SimpleTextEditor.frmEditor(this, caption);
   newForm.Show();       // Вывод формы
}
```

Прежде всего мы создаем новый заголовок. Переменная `nextFormNumber` описывается в самом начале класса следующим образом:

```
   private int nextFormNumber = 2;
```

Если бы мы решили использовать количество открытых окон в массиве форм (`MdiChildren`), то столкнулись бы с проблемой, если были открыты две формы, а затем форма с номером 1 закрыта. А так мы просто прибавляем единицу к текущему номеру каждый раз, когда открывается новая форма. Причина, по которой номер следующей формы равен 2, а не 1, заключается в том, что для инициализации первого окна в конструкторе используется текст `"Editor 1"`.



После этого мы создаем новый экземпляр редактора и выводим его.

Если запустить код и открыть новое окно, то можно будет заметить, что события в меню Edit относятся к нужному окну; это большая удача, поскольку мы не предпринимали никаких действий, для того, чтобы этого добиться. Поскольку для каждого экземпляра формы описываются свои обработчики событий, то сообщения о событиях каждый раз будут отправляться в определенное место. В списке Open Windows выведено два окна (см. рис. слева).

Теперь нужно закрыть открытые окна. Форма `frmContainer` обладает свойством `ActiveMdiChild`, которое позволяет идентифицировать то дочернее окно, которое мы хотим закрыть. Щелкните два раза мышью на меню Close и добавьте следующий код:

```
private void menuItemClose_Click(object sender, System.EventArgs e)
{
   // Определение активного дочернего MDI-окна
   SimpleTextEditor.frmEditor frm =
                          (SimpleTextEditor.frmEditor)this.ActiveMdiChild;
   if (frm != null)          // Перед тем как использовать потолок,
                             // необходимо убедиться в том, что он доступен
   {
      frm.Close();           // Закрытие окна
   }
}
```

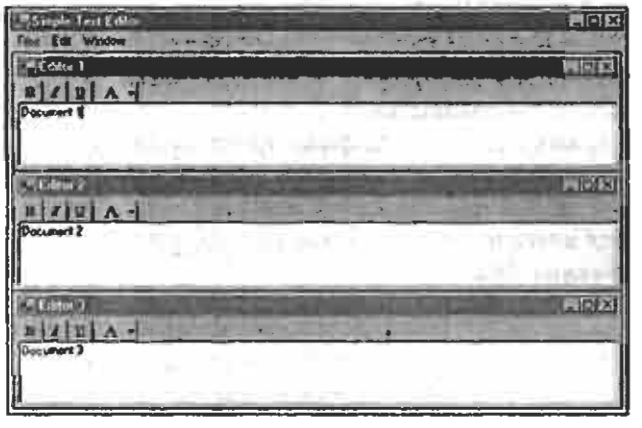
Сначала мы изменяем тип формы, содержащейся в свойстве `ActiveMdiChild` на класс `SimpleTextEditor`. Затем, прежде чем выполнять какие-либо действия с данным экземпляром, мы убеждаемся, что его значение не равно `null`, а потом вызываем метод `Close()` для данного окна.

Мы не будем разбирать реализацию других пунктов меню **Files**, поскольку пунктами **Open**, **Save** и **SaveAll** используются стандартные файловые диалоговые окна **Open** и **Save**, которые обсуждаются в следующей главе. Для пункта меню **Exit** используется тот же самый код, который встречался в предшествующих примерах. Весь код можно загрузить с сайта `www.wrox.com`.

Перейдем к двум оставшимся меню в меню **Window** — **Tile** и **Cascade**. Существует два возможных способа располагать вновь открываемые документы поверх уже открытых — вертикально и горизонтально; в данном примере мы будем располагать окна горизонтально. Для последовательного расположения нескольких окон с документами имеется только одна возможность. Щелкните два раза мышью на меню **Tile** и на меню **Cascade** и введите следующий код:

```
private void menuItemTile_Click(object sender, System.EventArgs e)
{
   this.LayoutMdi(MdiLayout.TileHorizonal);
}

private void menuItemCascade_Click(object sender, System.EventArgs e)
{
   this.LayoutMdi(MdiLayout.Cascade);
}
```



Метод `LayoutMdi` основного окна позволяет изменять порядок расположения всех дочерних MDI-окон.

Если запустить созданный код, то рисунок слева покажет, как выглядят три налагающихся друг на друга документа.

Создание управляющих элементов

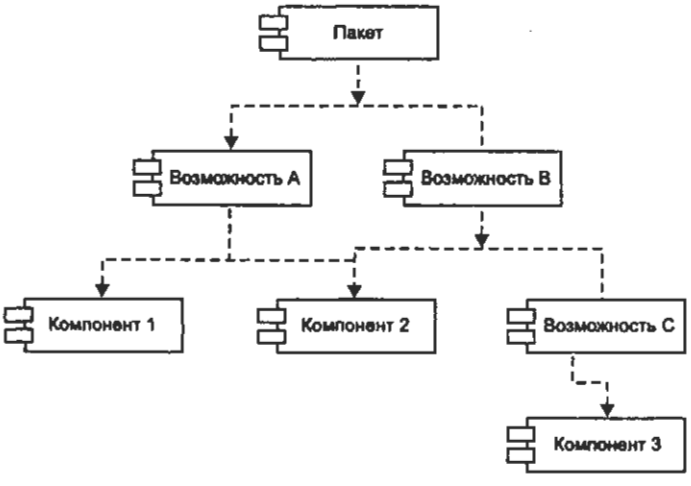
Вполне вероятна ситуация, при которой управляющие элементы, поставляемые совместно с Visual Studio.NET в готовом виде, не будут удовлетворять вашим требованиям. На то может существовать много причин: управляющие элементы имеют другой внешний вид, эти элементы налагают какие-либо ограничения, вам требуется включить в интерфейс управляющих элементов дополнительные возможности, или, наконец, необходимого вам управляющего элемента просто не существует. Признавая это, компания Microsoft предоставляет средства для создания элементов, которые удовлетворяли бы требованиям пользователя. В Visual Studio.NET имеется тип проекта Windows Control Library (библиотека управляющих элементов Windows), к помощи которого следует прибегнуть при необходимости создать свой собственный управляющий элемент.

Возможно разработать два различных вида собственных элементов, которые называются пользовательскими управляющими элементами (или составными управляющими элементами) и специализированными управляющими элементами (custom controls):

- *Пользовательские, или составные управляющие элементы*: Они строятся на основе функциональных возможностей уже существующих управляющих элементов, в результате чего создается новый управляющий элемент. Такие элементы обычно создаются либо для объединения функциональных возможностей с пользовательским интерфейсом управляющего элемента, либо для расширения возможностей интерфейса управляющего элемента путем объединения нескольких различных элементов в одну логическую единицу.
- *Специализированные управляющие элементы*: Эта разновидность управляющих элементов создается, когда ни один из существующих элементов не удовлетворяет вашим нуждам и управляющий элемент создается с нуля. Он воспроизводит весь пользовательский интерфейс, и при его создании не используются готовые управляющие элементы. Обычно необходимость в таких элементах возникает в тех случаях, когда пользовательский интерфейс создаваемого управляющего элемента существенно отличается от имеющихся в наличии.

В настоящей главе мы сосредоточим внимание на создании пользовательских управляющих элементов, поскольку вторая возможность — разработка и создание управляющих элементов с нуля — выходит за рамки данной книги. В главе 16, посвященной GDI+, представляются средства, необходимые для самостоятельного создания графических элементов. Познакомившись с GDI+, вы получите возможность без затруднений перейти к разработке специализированных управляющих элементов.

Управляющие элементы ActiveX (например, использующиеся в Visual Studio 6) находятся в специальном файле с расширением ocx. На самом деле, такой файл представляет собой COM DLL. В .NET управляющие элементы существуют в таком же виде, что и любой произвольный модуль, поэтому расширение ocx исчезло, а управляющие элементы существуют в виде обычных DLL.



Пользовательские управляющие элементы наследуются от класса `UserControl`. Это базовый класс предоставляет вновь создаваемому элементу все базовые возможности, которыми должен обладать управляющий элемент в .NET — на долю программиста остается только создание самого элемента. В качестве управляющего элемента может создаваться все, что угодно, начиная от надписи с каким-либо оригинальным дизайном и до занимающего весь экран табличного управляющего элемента (см. рис. слева).

В отличие от пользовательских элементов, специально создаваемые управляющие элементы обычно являются производными от класса `Control`, а не от класса `UserControl`.

При работе с управляющими элементами следует принимать ряд положений как должное, и если создаваемый вами элемент не отвечает им, то высока вероятность того, что такой элемент не будет использоваться другими пользователями. Вот эти критерии:

- Поведение управляющего элемента в период разработки должно как можно точнее соответствовать поведению этого управляющего элемента в период выполнения. Это означает, что если элемент состоит из надписи и текстового окна и называется `LabelTextbox`, то и надпись, и текстовое окно, и вводимый для надписи текст должны быть видимыми в процессе разработки. Хотя в приведенном выше примере это не представляет никаких трудностей, то в более сложных случаях это может создавать определенные проблемы, что потребует от вас находить приемлемый компромисс.
- Необходимо обеспечить логический доступ к свойствам управляющего элемента из формы, используемой для разработки. Хорошим примером здесь может служить элемент `ImageList`, который представляет диалоговое окно, позволяющее осуществлять поиск рисунков для включения в элемент, а также выводящее в диалоговом режиме список импортированных рисунков.

Ниже рассматривается процесс создания управляющих элементов на конкретном примере. Мы создадим элемент `LabelTextbox` и продемонстрируем основы создания проекта по разработке пользовательского управляющего элемента, его свойств и событий и отладке управляющего элемента.

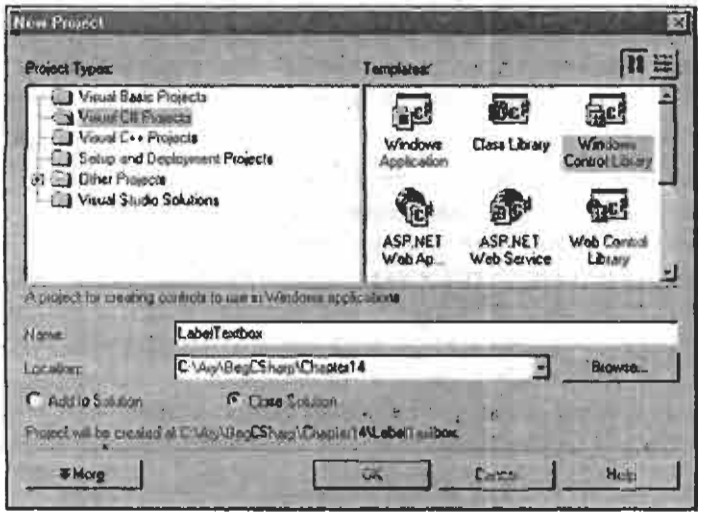
Практикум: пример создания управляющего элемента LabelTextbox

Как следует из самого названия элемента, для создания нового элемента в нем используются два уже существующих, что позволяет за один шаг максимально просто решить задачу, которая довольно часто возникает при программировании в Windows: добавить к форме надпись, далее добавить к той же самой форме текстовое окно, а затем определить местоположение этого окна относительно надписи. Посмотрим, чего может ожидать от такого элемента пользователь:

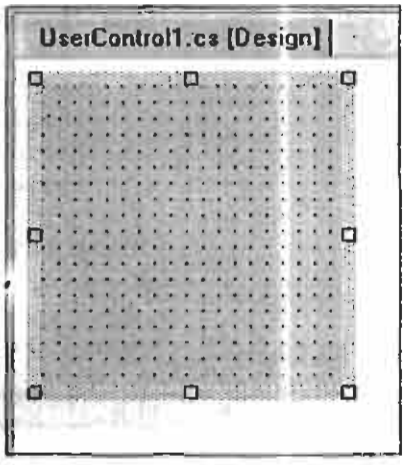
- Возможность размещать текстовое окно как справа от надписи, так и под надписью. Если текстовое окно располагается справа от надписи, то должна быть возможность задавать фиксированное расстояние между левым краем управляющего элемента и текстовым окном, для того чтобы расположить несколько текстовых окон одно под другим.
- Обычные свойства и событий текстового окна и надписи, которые доступны в обычном режиме.

Поставленная задача понятна — самое время запустить Visual Studio и создать новый проект:

1. Создайте новый проект C# в Visual Studio. Выберите проект с названием Windows Control Library. Назовите вновь создаваемый проект `LabelTextbox` (см. след. рис.).



Программа разработки форм представляет рабочую поверхность, которая несколько отличается используемой до сих пор. Во-первых, ее площадь оказывается существенно меньше, чем обычно, а, во-вторых, она вообще не выглядит как диалоговое окно. Это не должно обескураживать вас никоим образом — все работает в нормальном режиме. Основное отличие заключается в том, что там, где мы до сих пор размещали управляющие элементы на форме,



теперь будем создавать управляющий элемент, чтобы затем поместнть его на форму (см. рис. справа).

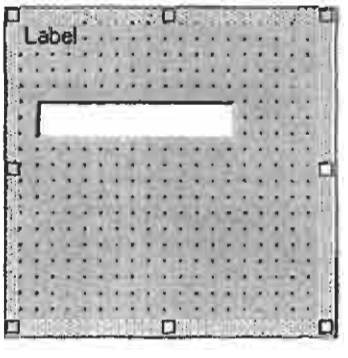
2. Щелкните мышью на рабочей поверхности, и вы увидите свойства управляющего элемента. Измените имя элемента на `ctlLabelTextBox`.

3. Щелкните два раза мышью на надписи в окне с инструментами и перенесите ее в форму. Измените ее имя на `lblTextbox`. Присвойте свойству `Text` значение `Label`. Присвойте свойству `AutoSize` значение `true`. Это приведет к тому, что размеры надписи будут определяться в соответствии с размерами текста.

4. Щелкните два раза мышью на элементе `Textbox` в окне с инструментами. Измените его свойство `Name` на `txtLabelText` и очистите свойство `Text`.

На этапе разработки нам не известно, каким образом пользователь пожелает расположить эти управляющие элементы. Поэтому нужно написать код, который будет отвечать за расположение надписи и текстового окна. Этот же код будет определять местоположение управляющих элемеитов и тогда, когда элемент `LabelTextbox` будет помещен в форму.

Структура управляющего элемента выглядит пока весьма разочаровывающе — `Textbox` свободно перемещается по рабочей поверхности, а сама поверхность слишком велика. На самом деле, это не имеет значения, поскольку, то, что вы видите, это совсем не то, что вы получите в итоге! Код, который мы сейчас добавим к управляющему элементу, изменит его внешний вид, но только после добавления элемента к форме (см. рис. слева).



Сначала следует определить взаимное расположение управляющих элементов друг относительно друга. Поскольку пользователь должен самостоятельно принимать решение об этом расположении, мы добавим элементу не одно, а два свойства. Одно из них будет называться `Position` (положение) — оно предоставляет пользователю возможность выбора: `Right` (справа) и `Below` (внизу). Если пользователь выбирает значение `Right`, то в дело вступает второе свойство. Оно

называется TextBoxMargin (поля текстового окна) и представляет собой целое число, соответствующее числу пикселей между левым краем управляющего элемента и тем местом, в котором должно быть размещено текстовое окно. Если пользователь задает значение, равное 0, то текстовое окно размещается так, чтобы его правый край располагался вровень с правым краем управляющего элемента.

Добавление свойств

Для того чтобы пользователь мог выбирать между Right и Below, прежде всего потребуется перечислимый тип с этими значениями. Вернитесь в проект по созданию управляющего элемента, перейдите в редактор кода и введите следующий код:

```
public class ctlLabelTextbox : System.Windows.Forms.UserControl
{
   //Нумерация двух возможных позиций
   public enum PositionEnum
   {
      Right,
      Below
   }
```

Это всего лишь обычный перечислимый тип, с которым мы познакомились в главе 5. Теперь о необычном — нам требуется, чтобы местоположение представляло собой свойство, которое может задаваться пользователем как программным путем, так и в процессе разработки. Этого можно добиться, включив это свойство в класс ctlLabelTextBox. Сначала, однако, потребуется создать два поля, в которых будут храниться значения, выбранные пользователем:

```
// Поле - элемент данного класса, в котором будет храниться
// информация о выборе, сделанном пользователем
private PositionEnum mPosition = PositionEnum.Right;
private int mTextboxMargin = 0;

public ctlLabelTextbox()
{
   ...
```

Затем мы добавляем свойство Position:

```
public PositionEnum Position
{
   get
   {
      return mPosition;
   }
   set
   {
      mPosition = value;
      MoveControls();
   }
}
```

Это свойство добавлено в класс точно так же, как и любое другое. Если возникнет необходимость вернуть его значение, мы вернем поле mPosition, а если потребуется изменить свойство Position — присвоим полю mPosition соответствующее значение и вызовем метод MoveControls(). К методу MoveControls() мы вернемся чуть ниже, а пока достаточно просто знать, что этот метод располагает два управляющего элемента в соответствии со значениями свойств mPosition и mTexBoxMargin.

Аналогично создается свойство `TexBoxMargin` за исключением того, что оно работает с целыми числами:

```
public int TextboxMargin
{
   get
   {
      return mTextboxMargin;
   }
   set
   {
      mTextboxMargin = value;
      MoveControls();
   }
}
```

Добавление обработчиков событий

Прежде чем перейти к тестированию этих двух свойств, необходимо добавить два обработчика событий. Когда управляющий элемент помещается в форму, вызывается событие `Load`. Это событие следует использовать для инициализации управляющего элемента и всех ресурсов, которые могут им использоваться. Мы обрабатываем это событие, для того чтобы перемещать управляющие элементы и подгонять размер всего управляющего элемента под размер содержащихся в нем элементов. Мы будем обрабатывать еще одно событие — `SizeChanged`. Оно возникает всегда, когда изменяются размеры управляющего элемента, и необходимо обрабатывать это событие, чтобы управляющий элемент изображался надлежащим образом. Мы подписываемся на события в конструкторе управляющего элемента:

```
// Обработка события SizeChanged
this.SizeChanged += new System.EventHandler(this.OnSizeChanged);

// Обработка события Load
this.Load += new EventHandler(this.OnLoad);
```

Затем добавляем обработчики событий:

```
private void OnLoad(object sender, EventArgs e)
{
   lblTextBox.Text = this.Name;          //Добавляет текст к надписи
   // Задание высоты управляющего элемента
   this.Height = txtLabelText.Height + lblTextBox.Height;
   MoveControls();
}

private void OnSizeChanged(object sender, System.EventArgs e)
{
   MoveControls();
}
```

Метод `MoveControls()` вызывается для изменения местоположения управляющих элементов. Следует рассмотреть этот метод, после чего протестировать работу управляющего элемента еще раз:

```
private void MoveControls()
{
   switch (mPosition)
   {
      case PositionEnum.Below:
         // Располагаем Textbox таким образом, чтобы его верхний
         // край находился непосредственно под надписью
```

```
            this.txtLabelText.Top = this.lblTextBox.Bottom;
            this.txtLabelText.Left = this.lblTextBox.Left;

            // Изменяем ширину Textbox, делая ее равной ширине
            // управляющего элемента
            this.txtLabelText.Width = this.Width;
            this.Height = txtLabelText.Height + lblTextBox.Height;
            break;
        case PositionEnum.Right:
            // Располагаем Textbox таким образом, чтобы его верхний
            // край совпадал с верхним краем надписи
            txtLabelText.Top = lblTextBox.Top;

            // Если поля обладают нулевым значением, то Textbox
            // располагается непосредственно рядом с надписью
            if (mTextboxMargin == 0)
            {
                int width = this.Width-lblTextBox.Width-3;
                txtLabelText.Left = lblTextBox.Right + 3;
                txtLabelText.Width = width;
            }
            else
            {
                // Если поля обладают ненулевым значением, то Textbox
                // располагается в месте, определенном пользователем
                txtLabelText.Left = mTextboxMargin;
                txtLabelText.Width = this.Right-mTextboxMargin;
            }
            break;
    }
}
```

Значение, хранящееся в `mPosition`, используется для определения того, следует ли располагать текстовое окно снизу или справа от надписи. Если пользователь выбирает вариант `Below`, то мы перемещаем текстовое окно таким образом, чтобы его верхняя граница совпала с нижней границей надписи. После этого мы перемещаем левый край текстового окна на уровень левого края всего управляющего элемента и устанавливаем его ширину, равной ширине управляющего элемента.

Если пользователь выбирает вариант `Right`, то в этом случае имеются две возможности. Если `TexBoxMargin` равно 0, то мы начинаем с того, что определяем ширину незанятой части управляющего элемента, предназначенной для размещения текстового окна. После этого мы устанавливаем левый край текстового окна так, чтобы он касался правого края текста, и задаем ему такую ширину, чтобы он занимал все имеющееся пространство. Если пользователь задаст поля, то мы вновь поместим левый край текстового окна в ту же самую позицию и заново определим его ширину.

Теперь можно протестировать управляющий элемент.

Отладка пользовательских управляющих элементов

Отладка пользовательских управляющих элементов существенным образом отличается от отладки приложений Windows. Обычно вы просто вставляете куда-либо в программу точку останова, нажимаете *F5* и смотрите, что произойдет. Если вы до сих пор не знакомы с процедурой отладки, то за подробными пояснениями вам следует обратиться к главе 7.

Средству управления требуется некоторый контейнер, в котором оно будет выводиться, и нам придется такой контейнер ему предоставить. Мы сделаем это, создав проект Windows Application.

Практикум: отладка пользовательских управляющих элементов

1. В открытом проекте `LabelTextbox` выберите **Files | New Project** и воспользуйтесь опцией для создания нового проекта Windows Application. Назовите этот проект `LabelTextboxTest`. Щелкните мышью на **Add to Solution**. Нажмите **OK**.

   В **Solution Explorer** открыто сразу два проекта. Название первого — `LabelTextbox` — выведено на чистой поверхности. Это означает, что если мы захотим запустить наше решение, то отладчик попытается использовать проект по созданию управляющего элемента в качестве стартового проекта. Это закончится неудачей, поскольку проект по созданию управляющего элемента не является независимым. Для того чтобы зафиксировать этот факт, щелкните правой кнопкой мыши на имени нового проекта — `LabelTextboxTest` и выберите пункт **Set as Sartup Project** (использовать в качестве стартового проекта). На этот раз, если вы запустите решение, приложение Windows начнет выполняться и никаких ошибок не возникнет.

2. Для того чтобы передать приложению Windows информацию о создаваемом нами управляющем элементе, необходимо включить ссылку на этот управляющий элемент. Щелкните правой кнопкой мыши на заголовке **References** в **Solution Explorer** под проектом `LabelTextboxTest` и выберите пункт **Add References...**.

3. Выберите вкладку с именем **Projects**. Раскроется список, в котором содержится запись о проекте по созданию управляющего элемента. Для того чтобы добавить ссылку на проект, щелкните мышью сначала на пункте **Select**, а затем на **OK**.

4. Добавьте к форме управляющий элемент `ctlLabelTextbox`, дважды щелкнув на нем мышью. Замечание: если при попытке дважды щелкнуть мышью на `ctlLabelTextbox` вы получите сообщение об ошибке, то попробуйте поступить следующим образом — щелкните правой кнопкой мыши на `LabelTextbox` в **Solution Explorer** и выберите пункт **Remove**. Повторите действия, предусмотренные шагом 3, только вместо выбора проекта с помощью вкладки **Projects** щелкните мышью на пункте **Browse** и найдите местонахождение DLL с именем `LabelTextbox.dll`. Выберите ее и нажмите **OK**.

5. Обратитесь к коду и найдите в методе `InitializeComponent()` следующую строку:

   ```
   this.MyControl = new LabelTextbox.ctlLabelTextbox();
   ```

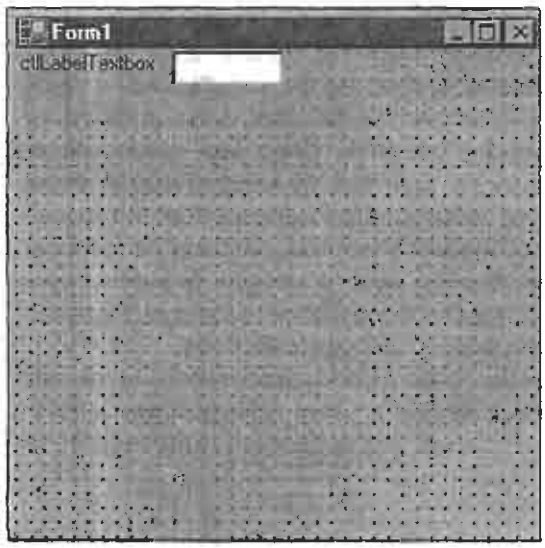
   Для этого, вероятно, придется развернуть область, помеченную как `"Windows Form Designer generated code"`.

6. Разместите на этой строке точку останова.

7. Запустите программу. Как и следовало ожидать, выполнение кода прервется в том месте, где помещена точка останова. Продолжите выполнение программы в пошаговом режиме (если вы используете клавиши, определяемыми по умолчанию, то этого можно добиться нажатием клавиши *F11*). Продолжая выполнять программу в пошаговом режиме, вы попадете в конструктор нового управляющего элемента — это как раз то, что необходимо для отладки компонента. Вы также можете расставить дополнительные точки останова. Для запуска приложения нажмите *F5*.

Расширение управляющего элемента LabelTextbox

Практикум: продолжение примера создания управляющего элемента LabelTextbox



Теперь можно протестировать свойства управляющего элемента. Повторите все шаги по созданию проекта в новом экземпляре Visual Studio, в котором существует управляющий элемент `LabelTextbox`. Затем перенесите управляющий элемент в форму. Надпись с именем управляющего элемента и текстовое окно занимают всю оставшуюся площадь управляющего элемента. Обратите внимание на то, что управляющие элементы, расположенные внутри управляющего элемента `LabelTextbox`, занимают правильные позиции, когда управляющий элемент добавляется к форме (см. рис. слева).

Добавление новых свойств

На данном этапе практически ничего нельзя делать с помощью названного управляющего элемента, поскольку у нас отсутствует возможность изменять текст надписи и текст в окне. Для работы с этим текстом мы добавим два свойства: `LabelText` и `TextBoxText`. Эти свойства добавляются точно так же, как и два предыдущих свойства:

```
public string LabelText
{
   get
   {
      return lblTextBox.Text;
   }
   set
   {
      lblTextBox.Text = value;
      MoveControls();    // Метод MoveControls вызывается для изменения
                         //размеров Textbox при наличии такой необходимости
   }
}

public string TextboxText
{
   get
   {
      return txtLabelText.Text;
   }
```

```
        set
        {
            txtLabelText.Text = value;
        }
    }
```

Если возникает необходимость включить некоторый текст, то мы просто присваиваем соответствующий текст надписи и свойству `Text` текстового окна и возвращаем значение названных свойств. Если текст надписи будет изменен, нам потребуется обратиться к методу `MoveControls()`, поскольку этот текст может оказывать влияние на то, каким образом будет располагаться текстовое окно. С другой стороны, текст, включаемый для вывода в текстовом окне, не приводит к перемещению управляющих элемеитов, и в том случае, если он окажется длиннее, чем текстовое окно, то он просто перестанет быть видимым.

Добавление новых обработчиков событий

Поскольку данный управляющий элемент является производным от класса `UserControl`, он наследует от него большое количество функциональных возможностей. Существует, одиако, определенное количество событий, которые мы не желаем передавать пользователю стандартным способом. Среди них можно назвать такие события, как `KeyDown`, `KeyPress` и `KeyUp`. Причина, по которой нам необходимо внести определенные изменения в эти события, заключается в том, что пользователь предполагает, что они будут отправляться в тот момент, когда он нажимает какую-либо клавишу в текстовом окие. А в том виде, в котором они существуют сейчас, эти события будут отправляться только в том случае, если сам управляющий элемент находится в фокусе и пользователь иажимает клавишу.

Для того чтобы изменить такое поведение, мы должны самостоятельно обрабатывать события, отправляемые текстовым окном, а затем переправлять их далее пользователю. Сначала нам необходимо подписаться на соответствующие события в конструкторе формы:

```
// Обработка события SizeChanged
this.SizeChanged += new System.EventHandler(this.OnSizeChanged);

// Обработка событий в Textbox, связанных с клавиатурой
this.txtLabelText.KeyDown += new KeyEventHandler(this.txtLabelText_KeyDown);
this.txtLabelText.KeyUp += new KeyEventHandler(this.txtLabelText_KeyUp);
this.txtLabelText.KeyPress += new
                        KeyPressEventHandler(this.txtLabelText_KeyPress);
```

Затем следует добавить трн обработчика событий:

```
private void txtLabelText_KeyDown(object sender, KeyEventArgs e)
{
    OnKeyDown(e);
}

private void txtLabelText_KeyUp(object sender, KeyEventArgs e)
{
    OnKeyUp(e);
}

private void txtLabelText_KeyPress(object sender, KeyPressEventArgs e)
{
    OnKeyPress(e);
}
```

Вызов метода `OnKey`*XXX* формирует обращение к любому методу, который подписан на данное событие.

Добавление специально созданного обработчика событий

Если требуется создать событие, которого не существует ни в одном из базовых классов, то придется выполнить несколько более серьезную работу. Мы создадим событие с именем `PositionChanged` (позиция изменена), которое будет возникать, когда изменяется значение свойства `Position`.

Для создания такого события потребуется три вещи:

- Нам будет необходим соответствующий делегат, который можно использовать для вызова методов, впоследствии привязываемых пользователем к событию.
- Пользователь должен обладать возможностью подписаться на событие посредством привязки к нему некоторого метода.
- Мы должны вызвать метод, который пользователь привязал к данному событию.

В данном случае мы будем использовать делегат `EventHandler`, предоставляемый нам системой .NET Framework. Это делегат особого вида, который объявляется с помощью своего собственного ключевого слова — `event`. Следующий код объявляет событие и предоставляет пользователю возможность на него подписаться:

```
public event System.EventHandler PositionChanged;

// Конструктор
public ctlLabelTexbox()
{
```

Теперь осталось только сгенерировать событие. Поскольку оно должно возникать в момент изменения значения свойства `Position`, мы генерируем событие в описателе доступа `set` к свойству `Position`:

```
public PositionEnum Position
{
   get
   {
      return mPosition;
   }
   set
   {
      mPosition = value;
      MoveControls();
      if (PositionChanged != null)          // Убеждаемся в наличии подписчиков
      {
         // Получаем список методов, которые необходимо вызвать
         System.Delegate[] subscribers = PositionChanged.GetInvocationList();
         // Проходим в цикле по всем методам
         foreach (System.EventHandler target in subscribers)
         {
            target(this, new EventArgs());       // Вызываем очередной метод
         }
      }
   }
}
```

Сначала нужно убедиться в наличии каких-либо подписчиков на данное событие, проверяя, не равняется ли значение `PositionChanged` `null`. Если нет, то мы вызываем метод `GetInvocationList`, который позволяет получить список делегатов,

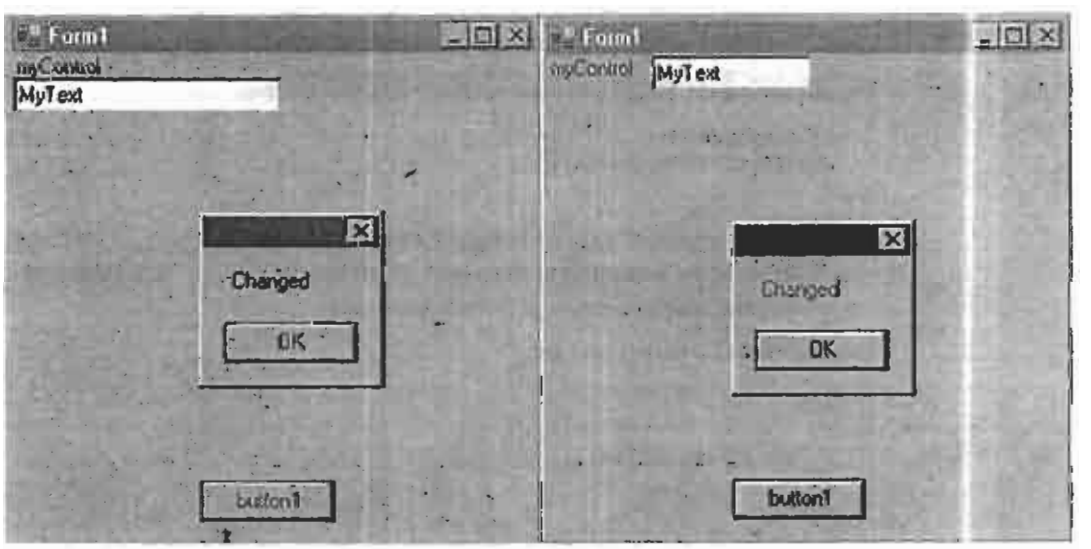
подлежащих вызову. После этого мы в цикле проходим по всем делегатам, вызывая каждый из них по очереди.

Подписка на созданное нами событие осуществляется в конструкторе проекта `LabelTextboxTest`:

```
public Form1()
{
   //
   // Требуется для поддержки Windows Form Designer
   //
   InitializeComponent();

   myControl.PositionChanged += new
                        EventHandler(this.myControl_PositionChanged);
}
}
```

Теперь можно откомпилировать и запустить тестовое приложение, содержащее созданный нами пользовательский управляющий элемент. Указанное приложение содержит также кнопку, которая изменяет внешний вид управляющего элемента, устанавливая и переустанавливая свойство `Position`, выводя при этом окно с сообщением.



Полный текст программ данного проекта может быть найден на web-сайте издательства Wrox.

Данный пример может быть несколько усовершенствован, но мы оставим это для вас в качестве упражнения.

Итоги

В настоящей главе мы начали с того, на чем остановились в предыдущей, — с изучения управляющих элементов `MainMenu` и `Toolbar`. Мы познакомились с тем, каким образом можно создавать MDI- и SDI-приложения и каким образом в этих приложениях могут использоваться меню и панели инструментов. Затем мы перешли к созданию своих собственных управляющих элементов, к разработке их свойств, пользовательского интерфейса и событий для таких управляющих элементов.

В следующей главе обсуждение Windows Forms будет завершено рассмотрением одной особой разновидности формы, которая известна нам пока только по названию — Windows Common Dialog.

Упражнение

1. Используя пример `LabelTextbox` в качестве основы, создайте новое свойство с нменем `MaxLength` (максимальная длина) для хранения максимально допустимого количества символов, которые могут находиться в текстовом окне. Затем создайте два новых события с нменами `MaxLengthChanged` (максимальная длина изменена) и `MaxLengthReached` (максимальная длина достигнута). Событие `MaxLengthChanged` должно возникать при изменении значения свойства, а событие `MaxLengthReached` — в тот момент, когда пользователь вводит символ, с учетом которого длина текста в текстовом окне становится равной значению `MaxLength`.

15

Использование диалоговых окон

В предыдущей главе мы узнали, каким образом можно реализовывать меню, панели инструментов, SDI- и MDI-формы и т. д. Кроме того, мы знаем, как создавать простые диалоговые окна, чтобы получать информацию от пользователя. Мы также создали достаточно сложные диалоговые окна, чтобы запрашивать у пользователя более специфическую информацию. Однако для выполнения стандартных операций, таких как открытие и сохранение файлов, вывод на печать, рекомендуется использовать заранее созданный класс диалоговых окон. В этом случае не только уменьшается объем кода, но и используются стандартные, хорошо всем знакомые диалоговые окна Windows. В .NET Framework имеются классы, которые позволяют использовать диалоговые окна Windows для открытия и сохранения файлов, для вывода на принтер, а также для выбора цвета и шрифта. Их использование позволяет пользователю не изучать сложную методологию, что было бы совершенно необходимым для кодирования таких функциональных возможностей с "нуля".

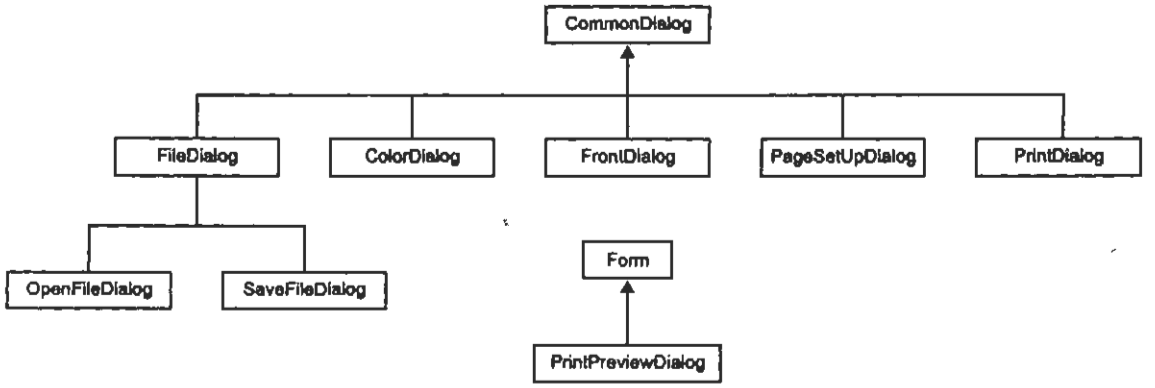
В данной главе мы познакомимся с тем, каким образом можно использовать стандартные классы диалоговых окон. Мы, в частности

- будем использовать классы `OpenFileDialog` и `SaveFileDialog`
- познакомимся с иерархией класса печати .NET и будем использовать классы `PrintDialog`, `PageSetupDialog` и `PrintPreviewDialog` для печати и предварительного просмотра печатаемой информации
- рассмотрим использование классов `FontDialog` и `ColorDialog` для изменения цвета и шрифта.

Диалоговые окна

Диалоговым называется окно, которое выводится в контексте другого окна. Мы можем обратиться к пользователю с просьбой ввести какие-либо данные, прежде чем продолжить выполнение программы. Общее диалоговое окно — это окно, которое часто используется для получения от пользователя наиболее распространенной информации. Оно является частью операционной системы Windows.

Имеющиеся в .NET Framework классы диалоговых окон представлены на следующем рисунке. Все они — за исключением класса `PrintPreviewDialog` — являются производными от абстрактного класса `CommonDialog` — базового класса, в котором представлены методы управления общим диалоговым окном в Windows.



Общие экземпляры методов и событий	Описание
`ShowDialog()`	Этот метод реализуется в производных классах для вывода общего диалогового окна.
`Reset()`	Каждый производный класс диалогового окна реализует метод `Reset()` для установки значений по умолчанию для всех свойств данного класса.
`HelpRequest`	Это событие генерируется в тот момент, когда пользователь щелкает мышью на кнопке Help общего диалогового окна.

В классе `CommonDialog` описаны методы и события (см. табл. слева), доступные любому классу общего диалогового окна.

Все эти классы построены на основе общего диалогового окна Windows с тем, чтобы сделать его доступным для приложений .NET. Исключением является окно `PrintPreviewDialog`, поскольку оно привносит свои собственные элементы в Windows Form, предназначенные для управления предварительным просмотром печати, и, следовательно, реально данное окно не является диалоговым. Классы `OpenFileDialog` и `SaveFileDialog` являются производными от базового абстрактного класса `FileDialog`, который добавляет общие файловые характеристики для диалоговых окон при открытии и сохранении файлов.

Эти диалоговые окна могут быть использованы:

- Для предоставления пользователю возможности искать и выбирать **файлы для открытия** используется диалоговое окно `OpenFileDialog`, которое может быть настроено так, чтобы открывать либо один, либо несколько файлов.
- С помощью диалогового окна `SaveFileDialog` пользователь может задавать имя **сохраняемого файла** и осуществлять поиск директории, в которой этот файл должен быть сохранен.
- Диалоговое окно `PrintDialog` используется для **выбора принтера** и задания опций, использующихся при печати.
- Диалоговое окно `PageSetupDialog` обычно используется для **определения параметров полей** страницы.
- Диалоговое окно `PrintPreviewDialog` позволяет осуществлять предварительный просмотр выводимой на печать информации на экране с предоставлением некоторых дополнительных возможностей, например увеличение просматриваемой информации.
- Диалог `FontDialog` выдает список всех инсталлированных в Windows **шрифтов** вместе со стилями и размерами, а также позволяет осуществлять предварительный просмотр выбранного шрифта.
- Класс `ColorDialog` упрощает выбор необходимого **цвета**.

Автору приходилось встречаться с приложениями, разработанными в одной и той же компании, в которых не предполагалось не только повторное использование стандартных общих диалоговых окон, но и какое-либо общее руководство по стилю создаваемых окон. Разработка таких диалоговых окон не приводила к получению стройного набора функциональных возможностей, а некоторые окна и кнопки располагались в различных местах, например, менялись местами кнопки Cancel и OK.

Иногда такая непоследовательность может быть обнаружена даже в рамках одного и того же приложения. Это препятствует нормальной работе пользователя и увеличивает время, необходимое для выполнения задачи.

Будьте последовательны при создании и использовании диалоговых окон! Этого можно добиться за счет применения общих диалоговых окон, а они используются во многих приложениях, в том числе и в тех, которые уже знакомы пользователю.

Использование диалоговых окон

Поскольку для всех классов диалога базовым классом является `CommonDialog`, то все классы диалога могут использоваться аналогичным образом. `ShowDialog()` и `Reset()` являются экземплярами общих методов. Метод `ShowDialog()` вызывает защищенный экземпляр метода `RunDialog()` для вывода диалога и по окончании работы возвращает экземпляр `DialogResult`, в котором содержится информация о том, какие действия предпринимал пользователь в процессе диалога. С другой стороны, метод `Reset()` присваивает всем свойствам диалога значения, использующиеся по умолчанию.

Приведенный ниже фрагмент кода демонстрирует пример использования класса диалога. Ниже каждый из этих шагов будет рассмотрен более подробно, а пока познакомимся с общей концепцией использования диалогов.

Из данного фрагмента программы вытекает, что:

- Сначала создается новый экземпляр класса диалога.
- Затем необходимо присвоить значения некоторым свойствам, для того чтобы разрешить/запретить использование дополнительных возможностей и определить состояние диалога. В данном случае свойству `Title` присваивается значение `"Sample"`, а флажку `ShowReadOnly` — `true`.
- После этого вызывается метод `ShowDialog()`, выводящий диалоговое окно, которое переходит в режим ожидания и реагирует на вводимую пользователем информацию.
- В тот момент, когда пользователь нажимает кнопку OK, диалоговое окно закрывается; проверка на нажатую кнопку OK осуществляется путем сравнения результата диалога со свойством `DialogResult.OK`. После этого можно получить доступ к значениям, введенным пользователем, обратившись к значению определенных свойств. В данном случае мы сохраняем значение свойства `FileName` в переменной `fileName`:

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.Title = "Sample";
dlg.ShowReadOnly = true;

if (dlg.ShowDialog() == DialogResult.OK)
{
   string fileName = dlg.FileName;
}
```

У каждого диалогового окна есть свои собственные настраиваемые возможности, которые будут рассматриваться в последующих разделах.

Использовать диалог в приложении Windows Forms оказывается даже проще, чем написать несколько строк кода. Программа создания Windows Forms сама генерирует код, ответственный за создание нового экземпляра и присваивание значений свойствам; нужно лишь вызвать метод `ShowDialog()`, а затем запросить измененные значения.

Диалоговые окна для работы с файлами

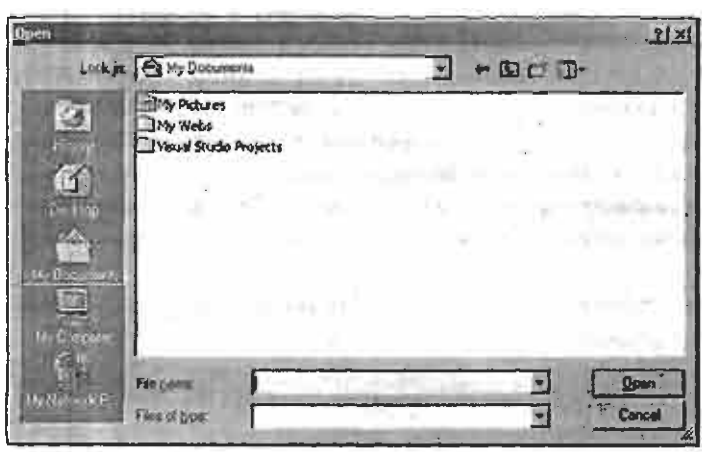
Диалоговые окна для работы с файлами позволяют пользователю выбрать соответствующее устройство и найти в файловой системе необходимый файл. Все, что для этого требуется,— получить от пользователя имя файла.

С помощью диалогового окна `OpenFileDialog` можно определить имя файла, который необходимо открыть, а используя окно `SaveFileDialog` — задать имя, под которым файл должен быть сохранен. Эти классы диалога очень похожи друг на друга, поскольку являются производными одного и того же абстрактного базового класса, хотя у каждого из этих классов имеется ряд присущих только одному из них свойств. В настоящем разделе мы сначала познакомимся с возможностями диалогового окна `OpenFileDialog`, а затем узнаем, чем отличается от него окно `SaveFileDialog`. Потом создадим пример приложения, в котором будут использованы оба этих диалоговых окна.

OpenFileDialog

Класс `OpenFileDialog` позволяет выбирать имя открываемого файла. Перед тем как вызывать метод `ShowDialog()`, сначала необходимо создать новый экземпляр класса `OpenFileDialog`.

```
OpenFileDialog dlg = new OpenFileDialog();
ShowDialog();
```



Выполнение программы, состоящей из этих двух строк кода, приведет к появлению на экране диалогового окна, представленного на рисунке слева.

Перед вызовом метода `ShowDialog()` есть возможность задать значения некоторых свойств, которые будут оказывать влияние на поведение и внешний вид диалогового окна либо налагать ограничения на файлы, которые могут быть открыты. В последующих разделах рассматриваются возможные модификации.

При применении OpenFileDialog в консольных приложениях необходимо использовать ссылку на модуль System.Windows.Forms и включить пространство имен System.Windows.Forms. Для приложений Windows Forms, которые созданы в Visual Studio.NET, это уже сделано программой-мастером в процессе создания.

Заголовок диалогового окна

В качестве заголовка окна OpenFileDialog по умолчанию используется слово Open (открытие). Заголовок диалога может быть изменен, для чего необходимо присвоить соответствующее значение свойству Title. Слово Open не всегда является самым лучшим вариантом названия, например для приложения, в котором необходимо проанализировать файлы регистрации с целью извлечения из них какой-либо информации или определения размера файла и которое закрывает файл непосредственно сразу после выполнения необходимых действий. В таких случаях файлы не остаются открытыми для пользователя, поэтому название вроде Analize Files (анализ файлов) может оказаться более подходящим. Так, например, в самой программе Visual Studio.NET есть несколько различных названий для диалогового окна открытия файла, чтобы дифференцировать типы открываемых файлов: Open Project (открытие проекта), Open File (открытие файла), Open Solution (открытие решения) и Open File from Web (открытие файла из Интернета).

Следующий фрагмент программы демонстрирует, каким образом можно присвоить диалоговому окну новый заголовок.

```
OpenFileDialog dlg = new OpenFileDialog();
dlg.Title = "Open File";
dlg.ShowDialog();
```

Задание директорий

По умолчанию диалог открывает директорию, которая открывалась пользователем, когда он в последний раз запускал приложение, и выводит хранящиеся в этой директории файлы. Это поведение может быть изменено посредством задания свойства InitialDirectory. По умолчанию в качестве значения свойства InitialDirectory используется пустая строка, которая соответствует директории пользователя My Documents, выводящейся при самом первом обращении к диалоговому окну в данном приложении. При втором обращении к диалоговому окну будет выводиться та же директория, в которой находится последний открывавшийся файл. Общее диалоговое окно Windows, которое используется окном OpenFileDialog, применяет реестр для хранения имени последнего открывавшегося файла.

Такое поведение может быть изменено присвоением свойства InitialDirectory строке, содержащей имя директории, перед обращением к методу ShowDialog(). В приложениях никогда не следует использовать строку с именем директории, "зашитую" в программу, поскольку такая директория в системе пользователя может отсутствовать. Для получения доступа к специальным системным папкам можно воспользоваться статическим методом GetFolderPath() класса System.Environment. Метод GetFolderPath() принимает параметр перечислимого типа Environment.SpecialFolder, в котором можно задать системную директорию, путь к которой необходимо получить.

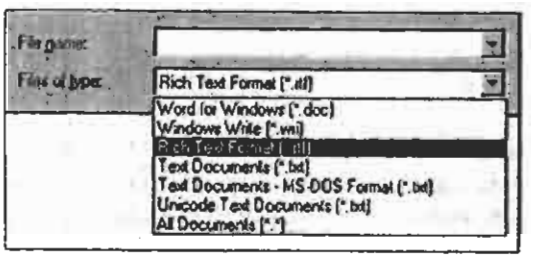
В следующем примере в качестве начальной директории используется обычная директория пользователя, предназначенная для хранения шаблонов:

```
string dir =
Environment.GetFolderPath(Environment.SpecialFolder.Templates);
dlg.InitialDirectory = dir;
```

Задание фильтра для файлов

Фильтр для файлов определяет типы файлов, которые пользователь может выбирать для открытия. Строка простого фильтра для файлов выглядит следующим образом:

```
Text Documents (*.txt)|*.txt|All Files|*.*
```



Этот фильтр используется для вывода вхождений в окне со списком Files of Type:. Microsoft WordPad выводит эти вхождения (см. рис. слева).

Фильтр состоит из нескольких отделов, разделенных вертикальной чертой (|). Он может состоять только из парных строк, поэтому число отделов всегда должно быть четным. Первая строка определяет текст, который будет выводиться в окне со списком; вторая строка используется для задания расширений файлов, которые будут выводиться в процессе диалога. Строке фильтра присваивается свойство `Filter`, как это показано в примере программы, приведенном ниже:

```
dlg.Filter = "Text Documents (*.txt)|*.txt|All Files|*.*";
```

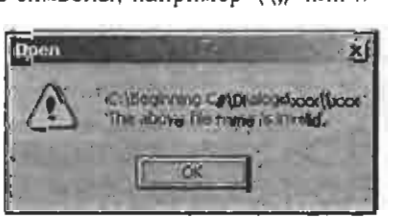
Присваивание неверного значения свойству `Filter` приведет к возникновению исключительной ситуации типа `System.ArgumentException`, при этом будет выдано следующее сообщение об ошибке: The provided filter string is invalid (Заданная строка фильтра является недопустимой). Пробелы перед или после строки фильтра также являются недопустимыми.

Свойство `FilterIndex` определяет выбор по умолчанию в окне со списком. В случае WordPad по умолчанию используется Rich Text Format (текстовый формат с богатыми возможностями), которому соответствует расширение `*.rtf`, как показано в выделенной строке на приведенном выше рисунке. Этот формат трудно читать, поэтому для просмотра следует открыть WordPad. Если есть возможность выбора из нескольких типов файлов, то можно присвоить свойство `FilterIndex` тому тип файла, который используется по умолчанию. Обратите внимание на то, что `FilterIndex` отсчитывается с 1!

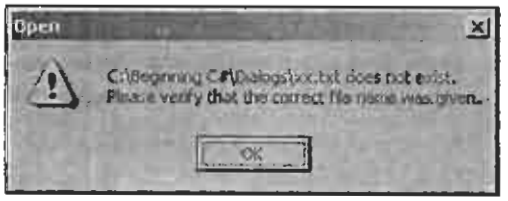
Проверка допустимости

В диалоговом окне `OpenFileDialog` могут автоматически выполняться определенные проверкн допустнмости файла, до того как вы попытаетесь его открыть. Имя файла, введенное пользователем, может быть проверено с помощью `ValidateNames` на предмет того, является ли оно допустнмым в Windows именем файла. Нажатне в диалоге кнопки OK с недопустимым именем файла приведет к выводу изображенного ниже сообщения, и пользователь будет лишен возможности выйти из диалога `OpenFileDialog`, не скорректировав имя файла. Недопустимое имя файла — это имя, в котором содержатся недопустимые символы, например \\,/ или :.

Ввод недопустимого имени файла приведет к появленню сообщения об ошибке: "The above file name is invalid." (Указанное выше имя файла недопустнмо.) (см. рис. справа).



Существуют два свойства, предназначенные для осуществления дополнительной проверки допустимости,— `CheckFileExists` и `CheckPathExists`. С помощью `CheckPathExists` проверяется допустимость заданного пути, а `CheckFileExists` проверяет наличие



заданного файла. Если указанного файла не существует, то при нажатии кнопки OK на экран будет выведено сообщение: "Please verify that the correct name was given." (Пожалуйста, убедитесь в том, что задано правильное имя файла.) (см. рис. слева).

По умолчанию все эти три свойства имеют значение `true`, то есть проверка допустимости выполняется автоматически.

Help

Диалоговое окно `OpenFileDialog` поддерживает использование кнопки Help, которая по умолчанию является невидимой. Присвоив свойству `ShowHelp` значение `true`, можно сделать кнопку Help видимой, кроме того, это даст возможность добавить обработчик событий `HelpRequest`, с помощью которого можно будет выводить вспомогательную информацию пользователю.

Результаты

Метод `ShowDialog()` класса `OpenFileDialog` возвращает результат в виде перечислимого типа `DialogResult`. В перечислимом типе `DialogResult` определены следующие допустимые значения: `Abort`, `Cancel`, `Ignore`, `No`, `None`, `OK`, `Retry` и `Yes`. `None` — это значение, используемое по умолчанию до тех пор, пока пользователь не закрыл диалог. Возвращаемый результат будет зависеть от того, какую кнопку нажал пользователь. В диалоговом окне `OpenFileDialog` могут быть возвращены только значения `DialogResult.OK` и `DialogResult.Cancel`.

Если пользователь нажал кнопку OK, то доступ к выбранному пользователем имени файла можно получить посредством свойства `FileName`. Если пользователь отменил диалог, то свойство `FileName` будет содержать пустую строку. Если значение свойства `MultiSelect` равно `true` (что означает, что пользователю предоставлена возможность выбирать более одного файла), то доступ ко всем выбранным файлам можно получить с помощью свойства `FileName`, которое в этом случае возвращает массив строк.

Небольшой фрагмент кода демонстрирует, каким образом можно извлечь несколько имен файлов из диалога `OpenFileDialog`:

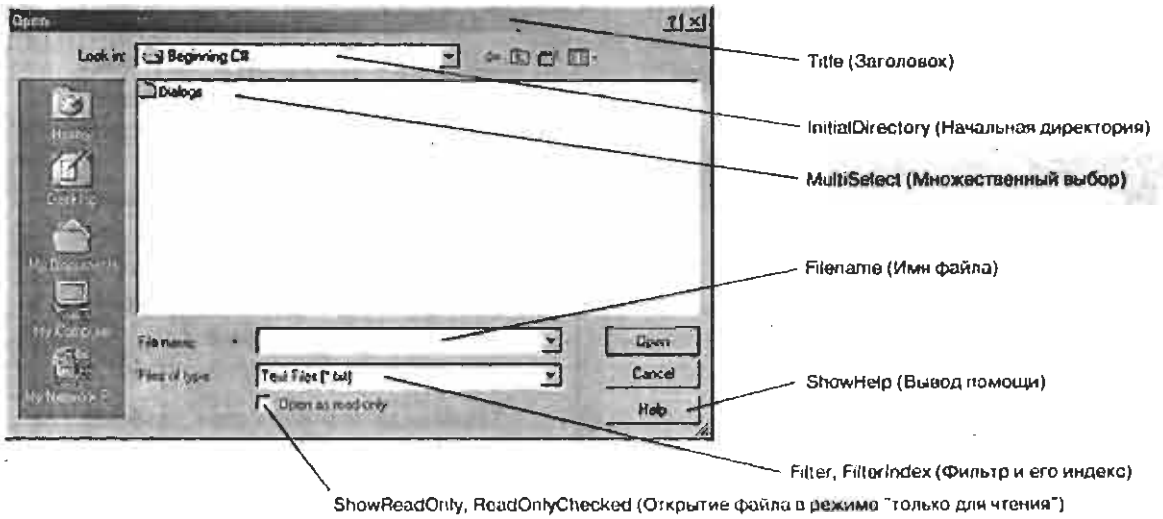
```
OpenFileDialog dlg = new OpenFileDialog();
dlg.MultiSelect = true;

if (dlg.ShowDialog() == DialogResult.OK)
{
   foreach (string s in dlg.FileNames)
   {
      Console.WriteLine(s);
   }
}
```

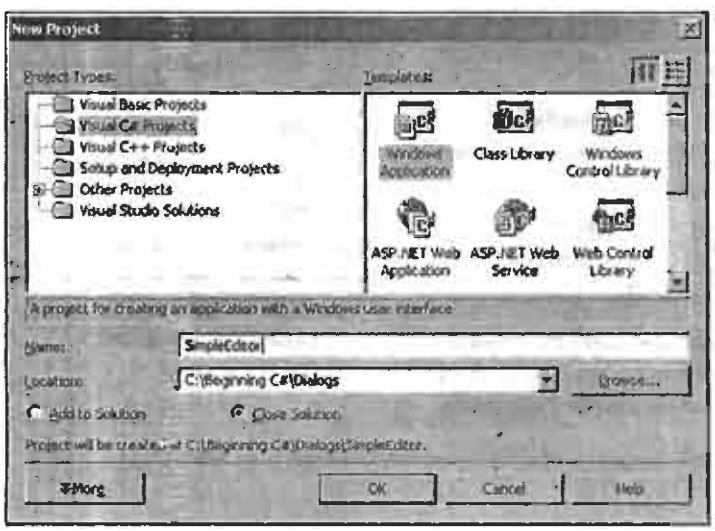
Метод `ShowDialog()` отрывает диалоговое окно. Свойству `MultiSelect` присваивается значение `true`, что означает, что у пользователя имеется возможность выбирать сразу несколько файлов. Нажатие кнопки OK приводит к завершению диалога, если все идет нормально, при этом возвращается значение `DialogResult.OK`. С помощью оператора `foreach` мы проходим по всем строкам, которые находятся в массиве строк, хранящемся в свойстве `FileName`, и выводим на экран каждый выбранный файл.

Свойства диалогового окна OpenFileDialog

На следующем рисунке показано диалоговое окно `OpenFileDialog`, изображенное со всеми возможными свойствами, разрешающими использование соответствующих элементов окна. По этому рисунку очень просто разобраться, какие свойства оказывают влияние на какие элементы пользовательского интерфейса:



Практикум: создание приложения Windows



1. Чтобы иметь возможность использовать диалоговые окна открытия и сохранения файлов, мы создадим простое приложение Windows `SimpleEditor` в окне, представленном слева.

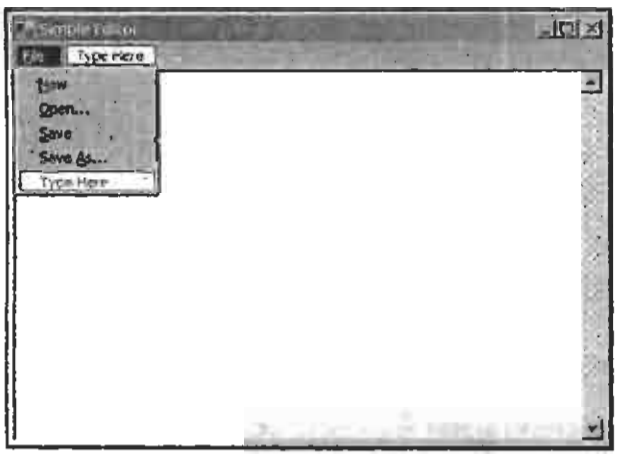
2. Переименуйте сгенерированную форму из `Form1.cs` в `SimpleForm.cs`, а класс `Form1` — в `SimpleEditorForm`. Кроме того, следует заменить пространство имен на `Wrox.Editor`.

   При изменении имени класса также необходимо изменить реализацию метода `Main()`, с тем чтобы учесть данное изменение, поскольку изменение значения свойства `Property` не влечет за собой автоматического изменения этого метода:

```
[STAThread]
static void Main()
{
   Application.Run(new SimpleEditorForm());
}
```

3. Присвойте свойству Text формы значение "Simple Editor". В качестве области, в которой будет осуществляться чтение и внесение изменений в файл, мы будем использовать многострочное окно, поэтому возьмите форму Textbox с панели инструментов и добавьте ее в программу разработки Windows Forms. Эта форма должна быть многострочной и занимать всю область, отведенную приложению, поэтому соответствующим свойствам следует присвоить значения, сведенные в таблицу справа.

Свойство	Значение
(Name)	textBoxEdit
Text	<пусто>
MultiLine	True
Dock	Fill
ScrollBars	Both
AcceptsReturn	True
AcceptsTab	True



4. Затем следует добавить в приложение MainMenu. В основном меню должно иметься вхождение File, в котором должны быть следующие субменю — New, Open, Save и Save As, как показано на рисунке слева.

Чтобы сделать код более удобным для чтения, замените имена на приведенные в нижеследующей таблице. Свойство Text будет выводиться в меню. Символ & указывает на то, что следующий за ним символ будет подчеркнут, и для обращения к данному элементу меню при использовании клавиатуры допускается одновременное нажатие подчеркнутого символа с клавишей <*Alt*>. Многоточие, присутствующее в свойстве Text вхождений Open и Save As, означает, что у пользователя до выполнения каких-либо действий будут запрашиваться дополнительные данные с помощью некоторого диалога. При обращении к меню File, New и Save все действия будут выполняться без какого-либо дополнительного вмешательства.

Название пункта меню	Текст
miFile	&File
miFileNew	&New
miFileOpen	&Open...
miFileSave	&Save
miFileSaveAs	Save &As...

Как только будет набран текст вхождения основного меню &File, под ним будет открыто субменю, в которое можно вводить его вхождения с помощью окна Type Here.

Почерпнуть дополнительную информацию о работе с меню можно в главе 14.

5. Обработчик для вхождения меню &New должен удалять все данные, находящиеся в текстовом окне посредством вызова метода Clear() управляющего элемента Textbox:

```
private void miFileNew_Click(object sender, System.EventArgs e)
{
    fileName = "Untitled";
    textBoxEdit.Clear();
}
```

6. Кроме того, переменная `fileName` должна получить значение `"Untitled"`. Необходимо объявить и инициализировать эту переменную в классе `SimpleEditorForm`:

```
private string fileName = "Untitled";
```

В меню **Open** нам потребуется выводить диалог открытия файлов, к созданию которого мы сейчас и приступим.

Практикум: считывание файла

Существует возможность передавать приложению `SimpleEditor` имя файла в качестве аргумента в момент запуска приложения. Передаваемое имя файла будет использоваться для открытия соответствующего файла и вывода его в текстовом окне.

1. Внесите следующие изменения в реализацию метода `Main()`, позволяющие передавать ему имя файла в качестве параметра:

```
[STAThread]
static void Main(string[] args)
{
   string fileName = null;
   if (args.Length != 0)
      fileName = args[0];
   Application.Run(new SimpleEditorForm(fileName));
}
```

2. Теперь необходимо изменить реализацию конструктора `SimpleEditorForm`, чтобы получить возможность использовать строку:

```
public SimpleEditorForm(string fileName)
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();

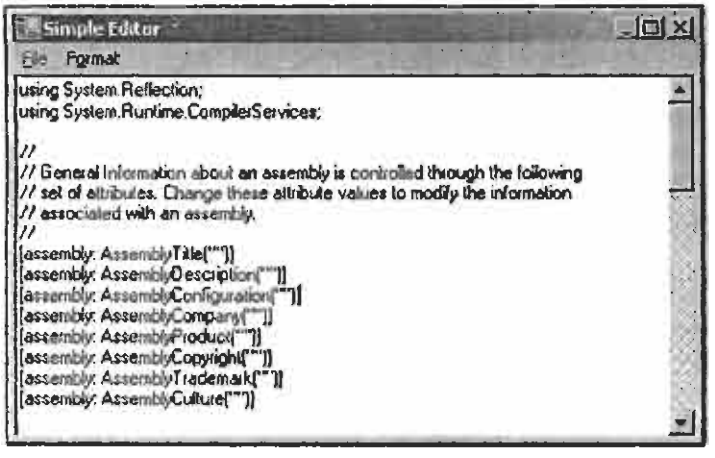
   if (fileName != null)
   {
      this.fileName = fileName;
      OpenFile();
   }
}
```

3. Нам также потребуется реализовать метод `OpenFile()`, который будет открывать файл и заполнять текстовое окно данными, взятыми из этого файла:

```
protected void OpenFile()
{
   try
   {
      using (StreamReader reader = File.OpenText(fileName))
      {
         textBoxEdit.Clear();
         textBoxEdit.Text = reader.ReadToEnd();
      }
   }
```

```
            catch (IOException ex)
            {
                MessageBox.Show(ex.Message, "Simple Editor",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
            }
        }
```

В данном случае для считывания файла используются классы `StreamReader` и `File`. Для того чтобы использовать эти классы и не указывать каждый раз пространство имен, в самое начало программы следует добавить оператор `using System.IO`.



4. Теперь можно запустить приложение из командной строки, передав ему имя файла. Этот файл будет немедленно открыт и выведен в текстовом окне, как это показано на рисунке слева.

Как это работает

Включив в число параметров метода `Main()` параметр `string[]`, мы получаем возможность использовать любые аргументы командной строки, задаваемые пользователем при запуске приложения:

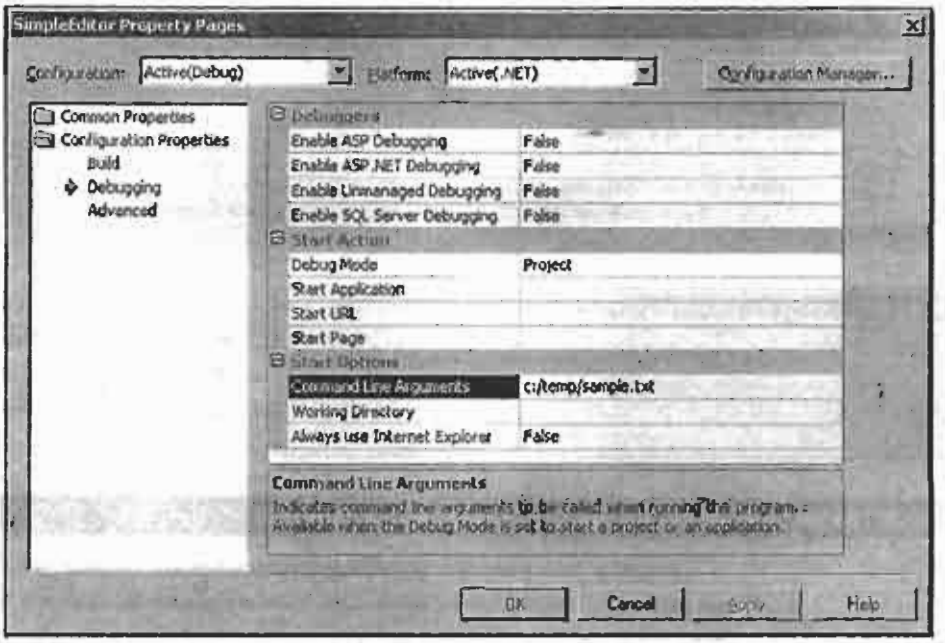
```
        static void Main(string[] args)
```

В методе `Main()` нами осуществляется проверка на предмет наличия переданных аргументов с помощью свойства `Length`. Если обнаруживается, что какие-либо аргументы были переданы, то первый аргумент присваивается переменной `fileName`, которая используется для передачи его конструктору формы `SimpleEditorForm`:

```
        {
            string fileName = null;
            if (args.Length != 0)
                fileName = args[0];
            Application.Run(new SimpleEditorForm(fileName));
        }
```

Как сказано в главе 7, в Visual Studio.NET существует также возможность описывать параметры командной строки для целей отладки. В Solution Explorer для этого следует просто выбрать проект, а затем обратиться к меню Project | Properties. Если вы выберете Configuration Properties | Debugging в левом дереве диалога, выведенном под раскрываемыми окнами, то вы сможете вводить аргументы командной строки (см. след. рис.).

В конструкторе `SimpleEditorForm` мы осуществляем проверку, обладает ли изначально переменная с именем файла каким-либо значением. Если да, то оно присваивается переменной `fileName`, после чего для открытия файла осуществляется вызов метода `OpenFile()`. Мы используем вызов отдельного метода `OpenFile()`,



а не выполняем открытие файла и заполнение текстового окна непосредственно в самом конструкторе класса, поскольку метод `OpenFile()` может быть использован и в других частях программы.

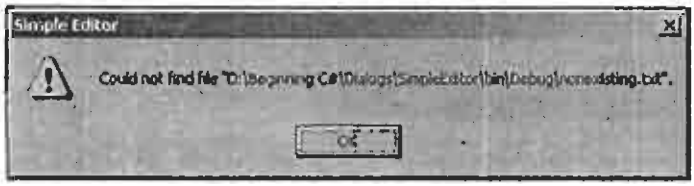
```
if (fileName != null)
{
   this.fileName = fileName;
   OpenFile();
}
```

В методе `OpenFile()` для открытия файла и получения возвращаемого класса `StreamReader` используется статический метод `OpenText()` класса `File`. Полученный класс `StreamReader` далее используется для чтения файла посредством метода `ReadToEnd()`, который загружает текст в виде строки и передает ее объекту `Textbox`. После завершения использования класса `StreamReader` его необходимо закрыть, для того чтобы освободить управляемые и неуправляемые ресурсы. Этого можно добиться с помощью оператора `using`, который вызывает метод `Dispose()` в конце блока, а метод `Dispose()`, реализованный в классе `StreamReader`, обращается к методу `Close()`, который осуществляет закрытие файла:

```
using (StreamReader reader = File.OpenText(fileName))
{
   textBoxEdit.Clear();
   textBoxEdit.Text = reader.ReadToEnd();
}
```

Поскольку выполнение операций с файлами запросто может приводить к возникновению исключительных ситуаций, например, если у пользователя отсутствуют необходимые права доступа к данному файлу или такого файла вообще не существует, то весь код заключается внутрь блока `try`. В случае возникновения какой-либо исключительной ситуации, связанной с вводом/выводом, на экран выводится сообщение, информирующее пользователя о возникшей проблеме, но работа самого приложения при этом не прерывается:

```
try
{
   // . . .
}
catch (IOException ex)
{
   MessageBox.Show(ex.Message, "Simple Editor",
         MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}
```



Если при запуске приложения будет введено несуществующее имя файла, то это приведет к появлению сообщения, представленного на рисунке слева.

Практикум: добавление диалога открытия файлов

Теперь у нас имеется возможность считывать файлы с помощью простого редактора, передавая имя файла при запуске приложения. Но мы предпочитаем использовать класс общего диалога — следующий класс, который мы собираемся добавить к приложению.

1. В окне с инструментами, которое используется в Windows Forms, можно обнаружить компонент `OpenFileDialog`. Добавьте этот компонент к программе разработки Windows Forms. При этом должны быть изменены всего лишь три свойства: в качестве имени экземпляра будет использоваться `dlgOpen`, свойство `Filter` присваивается приведенной ниже строке, а свойству `FilterIndex` в качестве значения присваивается 2, что приводит к выбору по умолчанию документов Wrox:

```
Text Documents (*.txt)|*.txt| Wrox Documents
(*.wroxtext)|*.wroxtext | All Files|*.*.
```

Как это работает

При добавлении программой разработки Windows Forms компонента `OpenFileDialog` в класс `SimpleEditorForm` включается новый частный элемент:

```
public class SimpleEditorForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox textBoxEdit;
   private System.Windows.Forms.MenuItem miFile;
   private System.Windows.Forms.MenuItem miFileNew;
   private System.Windows.Forms.MenuItem miFileOpen;
   private System.Windows.Forms.MenuItem miFileSave;
   private System.Windows.Forms.MenuItem miFileSaveAs;
   private System.Windows.Forms.MainMenu mainMenu;
   private System.Windows.Forms.OpenFileDialog dlgOpenFile;
```

В части кода, которая генерируется программой разработки Windows Forms, точнее, в методе `InitializeComponent()`, создается новый экземпляр класса `OpenFileDialog` и присваиваются значения указанным свойствам. Для того чтобы увидеть следующий код в редакторе, необходимо сначала щелкнуть мышью на символе +, расположенном в строке Windows Forms Designer generated code, а затем на символе +, в строке `private void InitializeComponent()`:

```
private void InitializeComponent()
{
   this.textBoxEdit = new System.Windows.Forms.TextBox();
   this.mainMenu = new System.Windows.Forms.MainMenu();
   this.miFile = new System.Windows.Forms.MenuItem();
   this.miFileNew = new System.Windows.Forms.MenuItem();
   this.miFileOpen = new System.Windows.Forms.MenuItem();
   this.miFileSave = new System.Windows.Forms.MenuItem();
   this.miFileSaveAs = new System.Windows.Forms.MenuItem();
   this.dlgOpenFile = new System.Windows.Forms.OpenFileDialog();
   // . . .
   //
   // dlgOpenFile
   //
   this.dlgOpenFile.Filter =
      "Text Documents (*.txt)|*.txt|Wrox Documents
(*.wroxtext)|*.wroxtext|All Files|*.*";
   this.dlgOpenFile.FilterIndex = 2;
```

С помощью программы разработки Windows Forms мы создали новый экземпляр класса `OpenFileDialog` и определили значения свойств. Теперь нам необходимо вывести диалог.

Практикум: вывод диалога OpenFileDialog

Добавьте обработчик событий "щелчок мышью" для вхождения меню **Open**, в котором мы будем выводить диалог и считывать выбранный файл, посредством следующего кода:

```
private void miFileOpen_Click(object sender, System.EventArgs e)
{
   if (dlgOpenFile.ShowDialog() == DialogResult.OK)
   {
      fileName = dlgOpenFile.FileName;
      OpenFile();
   }
}
```

Как это работает

Метод `ShowDialog()` выводит на экран диалог открытия файла и возвращает информацию о том, какая кнопка была нажата пользователем при выходе из него. Именно по этой причине мы производим сравнение с `DialogResult.OK` в операторе `if`. Если пользователь просто отменил диалог, мы не предпринимаем никаких действий:

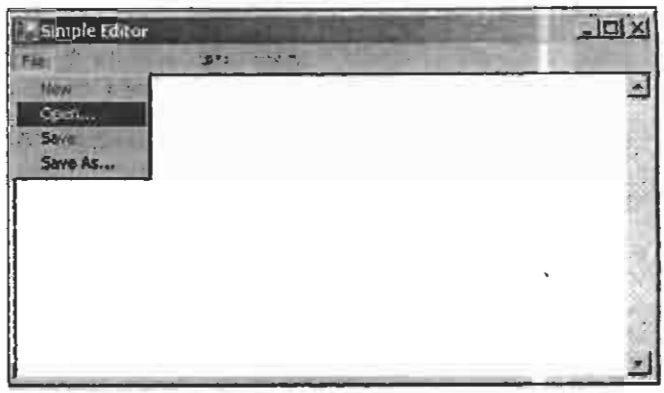
```
if (dlgOpenFile.ShowDialog() == DialogResult.OK)
{
```

Затем мы получаем выбранное имя файла, обратившись к свойству `FileName` класса `OpenFileDialog`, и присваиваем это значение переменной `fileName`. Это то самое значение, которое далее используется методом `OpenFile()`. Можно было бы открыть файл и непосредственно с помощью класса `OpenFileDialog`, осуществив вызов `dlgOpenFile.OpenFile()`, который также возвращает объект `Stream`, однако поскольку в нашем распоряжении уже имеется метод `OpenFile()`, который открывает и считывает файл, то мы используем именно его:

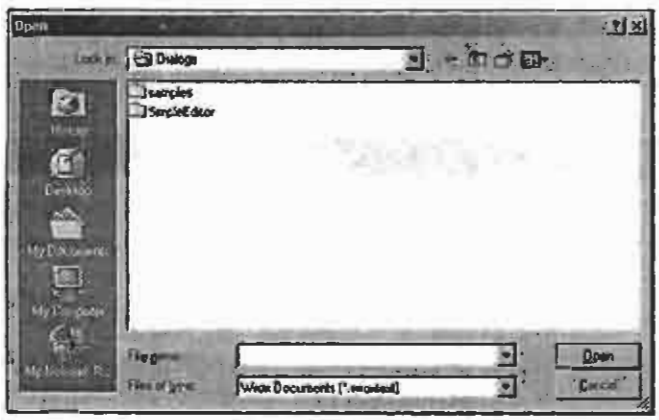
```
   fileName = dlgOpenFile.FileName;
   OpenFile();
```

Практикум: запуск простого редактора

1. Теперь есть возможность запустить простой редактор (см. рис. справа). На настоящий момент в нем будут функционировать только вхождения **New** и **Open...**. Вхождения **Save** и **Save As...** будут реализованы в следующем разделе.

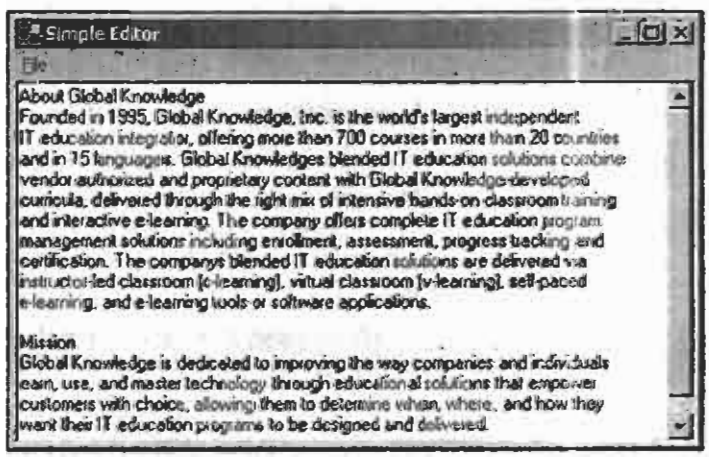


При выборе вхождения меню **File | Open** происходит вывод диалога `OpenFileDialog`, и мы получаем возможность выбрать необходимый файл. Мы исходим из предположения, что на вашем компьютере нет файлов с расширением `.wroxtext`. Мы пока еще не умеем сохранять файлы, поэтому для ведения диалога с редактором вы можете использовать файл с каким-либо другим расширением, либо предварительно скопировать произвольный текстовый файл в файл с расширением `.wroxtext`. Воспользуйтесь диалоговым окном, показанным на рисунке слева.



Выбрав текстовый файл и нажав кнопку **Open**, мы увидим его текст в текстовом окне диалога.

Автор в качестве примера выбрал расположенный на своей локальной системе файл `GlobalKnowledge.txt`, как показано на приведенном справа рисунке.



На настоящий момент мы научились считывать уже существующие файлы. Теперь стоит научиться создавать новые файлы и вносить изменения в уже существующие. Для решения данной задачи мы воспользуемся диалоговым окном `SaveFileDialog`.

Диалоговое окно SaveFileDialog

Классы `SaveFileDialog` и `OpenFileDialog` очень похожи и обладают целым набором одинаковых свойств. В настоящем разделе мы не будем рассматривать свойства, которые работают так же, как и для диалогового окна открытия файлов. Вместо этого мы сосредоточим внимание на свойствах, присущих только диалоговому окну сохранения файлов, или на тех общих для них свойствах, использование которых отлично.

Свойство Title

С помощью свойства `Title` (заголовок) можно задавать заголовок данного диалогового окна, подобно тому, как это делалось для окна `OpenFileDialog`. Если ничего не задавать, то будет выведено значение, использующееся по умолчанию,— Save As (сохранить как).

Расширения файлов

Расширения файлов используются для привязки файлов к приложениям. Наилучшая практика — обязательно добавлять расширение к имени файла, поскольку в противном случае Windows не будет в состоянии определить, какое именно приложение должно использоваться для открытия данного файла, да и вы сами, вероятнее всего, это вскоре забудете.

`AddExtension` — это свойство логического типа, которое определяет, должно ли расширение автоматически добавляться к имени файла, вводимого пользователем. Если пользователь самостоятельно ввел расширение файла, то в этом случае никаких дополнительных расширений добавляться не будет: если пользователь ввел в качестве имени файла `test`, то файл будет сохранен под именем `test.txt`. Если же пользователь ввел имя `test.txt`, то файл все рано будет сохранен под именем `test.txt`, а не под именем `test.txt.txt`.

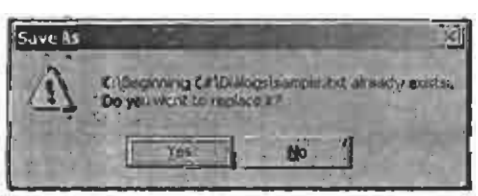
Свойство `DefaultExt` определяет расширение файла, используемое в тех случаях, когда пользователь не указывает расширения. Если это свойство остается пустым, то вместо него будет использоваться расширение, определяемое выбранным на данный момент значением свойства `Filter`. Если вы одновременно присвоите значения и свойству `DefaultExt`, и свойству `Filter`, то здесь будет использоваться значение свойства `DefaultExt`, независимо от значения свойства `Filter`.

Проверка допустимости

Для автоматического выполнения проверки допустимости имени файла существуют — как и для диалогового окна `OpenFileDialog` — свойства `ValidateNames`, `CheckFileExists` и `CheckPathExists`. Отличие между окнами `OpenFileDialog` и `SaveFileDialog` заключается в том, что в последнем по умолчанию используется значение свойства `CheckFileExists`, равное `false`, что означает, что существует возможиость задавать совершенно новое имя сохраняемому файлу.

Перезапись существующих файлов

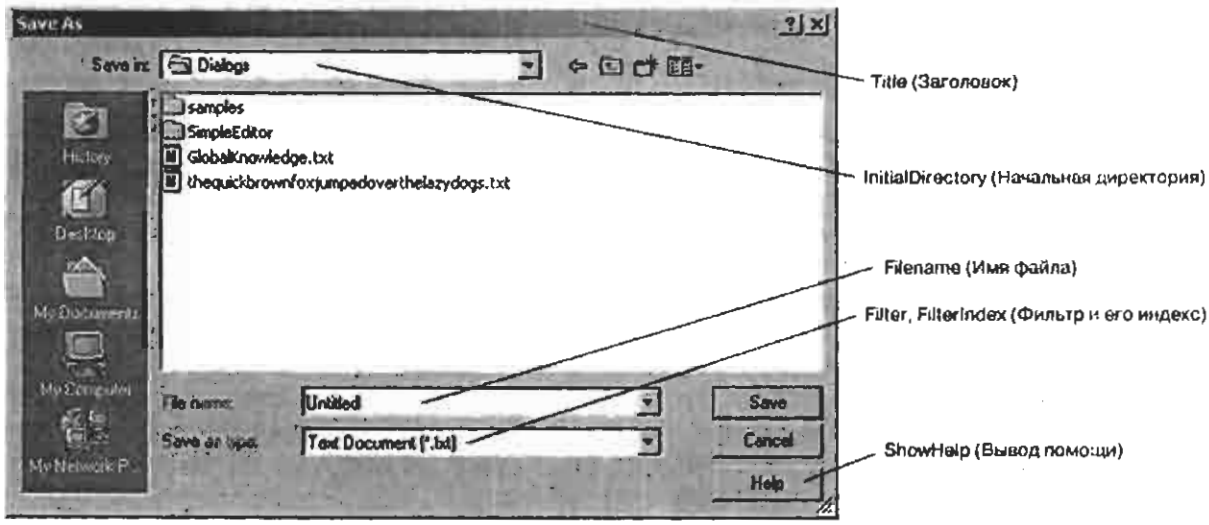
Проверка допустимости имен файлов аналогична проверке в `OpenFileDialog`. Однако в окне `SaveFileDialog` требуется выполнение некоторых дополнительных проверок и задание значений некоторых дополнительных свойств. Во-первых, если свойству `CreatePrompt` присвоено значение `true`, будет выведен запрос, должен ли создаваться новый файл. Если свойству `OverwritePrompt` присвоено значение `true`, то это означает, что будет выводиться запрос, действительно ли пользователь хочет осуществить запись на место уже существующего файла. По умолчанию свойству `OverwritePrompt` присваивается значение `true`, а свойству `CreatePrompt` — `false`.



При таких установках, если пользователь попытается сохранить уже существующий файл, будет выведено сообщение: C:\Beginning C#\Dialogs\sample.txt already exists. Do you want to replace it? (Файл с именем `C:\Beginning C#\Dialogs\sample.txt` уже существует. Вы действительно хотите его заменить?) (см. рис. слева).

Свойства SaveFileDialog

Ниже приведен рисунок, на котором сведены воедино и обозначены все свойства `SaveFileDialog`:



Практикум: добавление диалогового окна SaveFileDialog

1. Диалоговое окно `SaveFileDialog` можно добавить точно таким же образом, каким в программу разработки Windows Forms добавлялось окно `OpenFileDialog`: выберите компонент `SaveFileDialog` в окне с инструментами и перенесите его в форму. Измените его имя на `dlgSaveFile`, значение свойства `FileName` — на `Untitled`, значение свойства `FilterIndex` — на 2, а в качестве значения свойства `Filter` используйте приведенную ниже строку — точно так же, как это делалось для диалогового окна `OpenFileDialog`. Поскольку при записи файлов посредством данного редактора мы хотим разрешить использование только расширений `.txt` и `.wroxtext`, то элемент `*.*` в этой строке будет отсутствовать:

```
Text Documents (*.txt)|*.txt| Wrox Documents (*.wroxtext)|*.wroxtext.
```

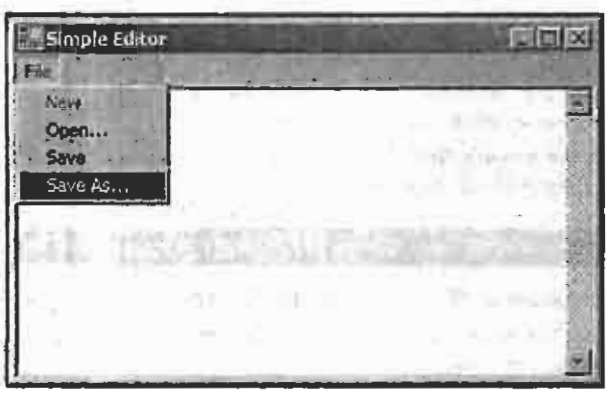
2. Добавьте обработчик событий `Click` вхождения меню Save As, а также код, приведенный ниже. Этот код предназначен для вывода диалога `SaveFileDialog` с помощью метода `ShowDialog()`. Результаты работы этого диалога будут интересовать нас только в том случае, если пользователь завершил диалог, нажав кнопку OK. Для сохранения файла на диске мы вызываем метод `SaveFile()`. Этот метод будет реализован на следующем шаге:

```
private void miFileSaveAs_Click(object sender, System.EventArgs e)
{
   if (dlgSaveFile.ShowDialog() == DialogResult.OK)
   {
      fileName = dlgSaveFile.FileName;
      SaveFile();
   }
}
```

3. Добавьте в файл метод `SaveFile()` в приведенном здесь виде:

```
protected void SaveFile()
{
   try
   {
      Stream stream = File.OpenWrite(fileName);
      using (StreamWriter writer = new StreamWriter(stream))
      {
         writer.Write(textBoxEdit.Text);
      }
   }
   catch (IOException ex)
   {
      MessageBox.Show(ex.Message, "Simple Editor",
         MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
   }
}
```

Как и в методе `OpenFile()`, для открытия файла используется класс `File`, однако в данном случае мы открываем его на запись с помощью метода `OpenWrite()`, возвращающего объект `Stream`, который передается конструктору класса `StreamWriter`. Метод `Write()` класса `StreamWriter` записывает в файл все данные, находящиеся в `textBox`. В конце блока `using StreamWriter` закрывается. Объект `Stream`, привязанный к `writer` также закрывается, поэтому никаких дополнительных обращений к методу `Close()` для объекта `Stream` не требуется.



Более подробно вы сможете познакомиться с классами, использующимися для осуществления ввода/вывода, в главе 20.

4. После того как проект будет построен, можно запустить приложение с помощью меню **Debug | Start** в **Visual Studio.NET**. Введите какой-либо текст в текстовое окно и перейдите в меню **File | Save As...**, как показано на рисунке слева.

Это приведет к появлению диалога `SaveFileDialog`, изображенного на рисунке справа. Теперь можно сохранить файл, а затем снова открыть его для внесения каких-либо дополнительных изменений.



Практикум: реализация обработчика событий Save

1. На настоящий момент мы можем выполнять операцию Save As, а вот простую операцию Save мы выполнять пока не умеем. Добавьте обработчик событий "щелчок мышью" во вхождение меню Save и введите следующий код:

```
private void miFileSave_Click(object sender, System.EventArgs e)
{
   if (fileName == "Untitled")
   {
      miFileSaveAs_Click(sender, e);
   }
   else
   {
      SaveFile();
   }
}
```

Как это работает

В меню Save файл должен сохраняться без открытия каких-либо дополнительных диалогов. Из этого правила есть единственное исключение — если создается новый документ, а пользователь не указал его имя, то обработчик Save должен работать как обработчик Save As и выводить диалог сохранения файла.

С помощью переменной `fileName` можно легко проверить, был ли данный файл открыт или имя файла по-прежнему имеет начальное значение `Untitled`, которое присваивается новому документу в момент его создания. Если в результате выполнения оператора `if` получается значение `true`, то мы просто вызываем обработчик `miFileSaveAs_Click()`, который был реализован нами ранее для меню Save As.

> *Если же файл был открыт, а пользователь выбирает меню Save, управление передается на блок* `else`. *В этом случае есть возможность использовать метод* `SaveFile()`, *разработанный ранее.*

Практикум: задание заголовка формы

В Notepad, Word и других приложениях Windows имя файла, который редактируется в настоящий момент, выводится в качестве заголовка данного приложения. Нам тоже следует реализовать такую возможность.

Создайте новый элемент — функцию `SetFormTitle()` и введите следующую реализацию:

```
protected void SetFormTitle()
{
   FileInfo fileinfo = new FileInfo(fileName);
      this.Text = fileinfo.Name + " - Simple Editor";
}
```

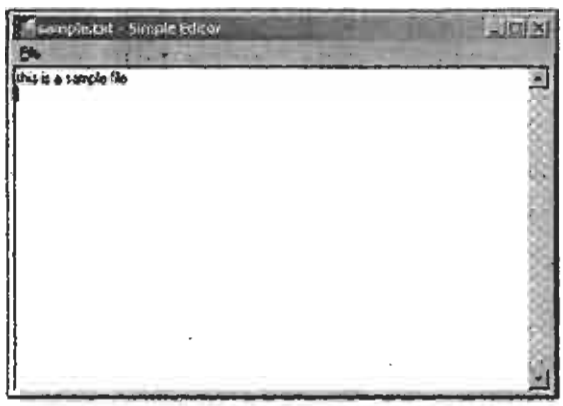
Класс `FileInfo` позволяет получить имя файла без предшествующего ему пути, который хранится в переменной `fileName`. Классы `FileInfo` и `StreamWriter` будут рассматриваться в главе 20.

В обработчики `miFileNew_Click()`, `miFileOpen_Click()` и `miFileSaveAs_Click()` добавьте обращение к этому методу сразу после присваивания значения переменной `fileName`, как показано во фрагментах кода, приведенных ниже:

```
private void miFileNew_Click(object sender, System.EventArgs e)
{
   fileName = "Untitled";
   SetFormTitle();
   textBoxEdit.Clear();
}

private void miFileOpen_Click(object sender, System.EventArgs e)
{
   if (dlgOpenFile.ShowDialog() == DialogResult.OK)
   {
      fileName = dlgOpenFile.FileName;
      SetFormTitle();
      OpenFile();
   }
}

private void miFileSaveAs_Click(object sender, System.EventArgs e)
{
   if (dlgSaveFile.ShowDialog() == DialogResult.OK)
   {
      fileName = dlgSaveFile.FileName;
      SetFormTitle();
      SaveFile();
   }
}
```



Как это работает

Каждый раз, когда происходит изменение имени файла, свойство `Text` используемой в этот момент формы будет изменяться на новое имя этого файла, к которому добавлено название приложения.

Запустив приложение, вы сможет увидеть на экран, подобный изображенному на рисунке слева. Поскольку в данном случае автор редактирует файл с именем `sample.txt`, то именно эта информация и выводится в качестве заголовка формы.

На этом создание простого редактора завершено: теперь он позволяет открывать, создавать и сохранять файлы. Можно ли этим ограничиться? Конечно, нет. Поскольку делопроизводство без использования бумаги пока еще не может существовать, нам потребуется реализовать некоторые возможности работы с принтером.

Вывод на печать

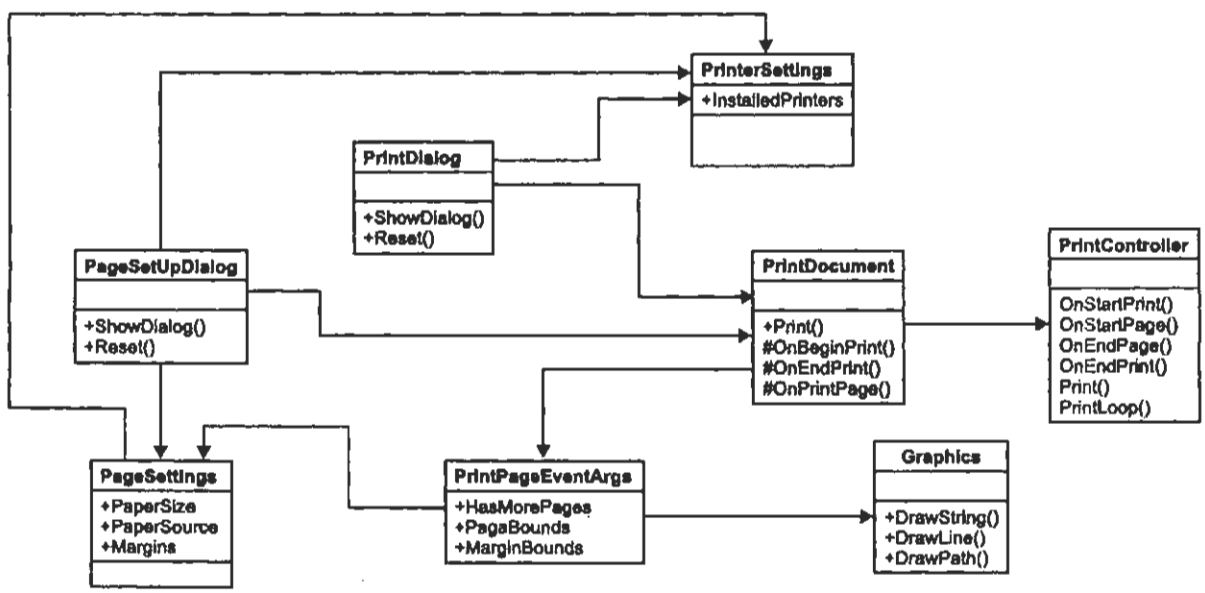
Когда дело доходит до печати, приходится думать о выборе принтера, установках вывода страницы и о том, как напечатать несколько страниц. Использование классов .NET, находящихся в пространстве имен `System.Drawing.Painting`, позволяет существенно облегчить нашу задачу.

Перед тем как перейти к рассмотрению класса `PrintDialog`, который позволяет выбирать принтер, необходимо кратко ознакомиться с тем, каким образом .NET управляет процессом вывода на печать. В основе вывода на печать лежит использование класса `PrintDocument`, обладающего методом `Print()`, который осуществляет целую последовательность вызовов, завершающуюся обращением к методу `OnPrintPage()` — методу, ответственному за передачу выходного потока принтеру.

Перед тем как перейти к более подробному рассмотрению того, каким образом устроен код, отвечающий за вывод на печать, следует посвятить еще некоторое время более детальному знакомству с классами .NET, связанными с выводом на печать.

Архитектура вывода на печать

Основные составные части архитектуры вывода на печать изображены на следующем рисунке в виде диаграммы, на которой показаны взаимоотношения между классами и некоторыми свойствами и методами:



Вот функциональные возможности этих классов:

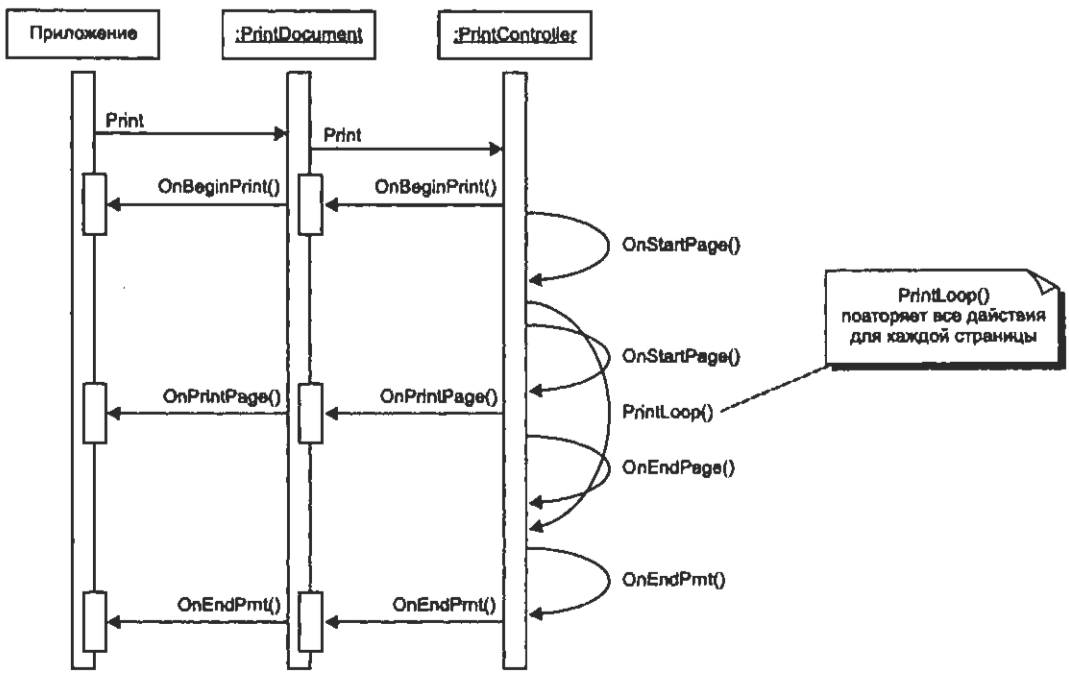
- Наиболее важный класс — `PrintDialog`. На приведенной диаграмме видно, что почти все классы связаны с ним. Для того чтобы вывести на печать какой-либо документ, необходим экземпляр класса `PrintDocument`. На следующей диаграмме мы увидим последовательность действий при выводе на печать, инициируемую данным классом.
- Класс `PrintController` осуществляет управление потоком задания вывода на печать. Контроллер вывода на печать запускает задачу вывода на печать, он обладает событиями для начала вывода на печать, для каждой страницы и для завершения вывода на печать. Это абстрактный класс, поскольку его реализации отличаются от реализации обычного вывода на печать и предварительного просмотра печатаемой информации.
- С помощью класса `PrinterSettings` имеется возможность определять и задавать значения свойств, определяющих конфигурацию принтера, например дуплексная печать, горизонтальное или вертикальное расположение листа, число копий и т. д.
- В задачу класса `PrintDialog` входит определение принтера, на который должна отправляться выводимая на печать информация,

и каким образом должны быть сконфигурированы установки в `PrinterSettings`. Этот класс является производным от класса `CommonDialog` — как и все остальные классы диалогов, с которыми мы уже успели познакомиться, поэтому у вас уже есть самое первое представление о том, каким образом он будет использоваться.

- Класс `PrinterSettings` определяет размеры и границы выводимых на печать страниц, а также то, какая требуется печать — цветная или черно-белая. Конфигурация этого класса может быть задана с помощью класса `PageSetupDialog`, который также является производным от `CommonDialog`.

Последовательность действий при выводе на печать

Теперь нужно рассмотреть основную последовательность действий при выводе на печать. На приведенной ниже диаграмме изображены основные элементы — приложение, которое нам предстоит создать, экземпляр класса `PrintDocument` и `PrintController` во временной последовательности:



Приложению необходимо обратиться к методу `Print` класса `PrintDocument`. В результате запускается выполнение последовательности действий для вывода на печать. Поскольку сам по себе экземпляр класса `PrintDocument` не является ответственным за процесс вывода на печать, эта работа возлагается на `PrintController` посредством вызова метода `Print()` данного класса.

Теперь в дело вступает контроллер печати — он передает экземпляру класса `PrintDocument` информацию о начале печати, обращаясь к методу `OnBeginPrint()`. Если приложению необходимо выполнить какие-либо действия в момент начала печати, то следует зарегистрировать обработчик событий в `PrintDocument` — это позволит приложению узнать о начале печати. На представленной диаграмме

предполагается, что обработчик OnBeginPrint() зарегистрирован, поэтому он может быть вызван из класса PrintDocument.

После завершения начального этапа PrintController обращается к PrintLoop(), в котором вызывается метод OnPrintPage() класса PrintDocument для каждой страницы, предназначенной для вывода на печать. Метод OnPrintPage() вызывает все обработчики событий PrintPage. Такой обработчик необходимо реализовывать всегда, поскольку в противном случае на печать не будет выведено никакой информации. Обработчик с именем OnPrintPage() показан на диаграмме.

После завершения печати последней страницы PrintController вызывает метод OnEndPrint() класса PrintDocument. Существует дополнительная возможность реализовать обработчик, вызываемый в данный момент.

Наиболее важную информацию можно сформулировать следующим образом:

Можно реализовать код, ответственный за вывод на печать, в обработчике событий PrintDocument.PrintPage. Он будет вызываться для каждой страницы, подлежащей выводу на печать. Если необходимо использовать код, который будет вызываться только один раз, то для этого требуется реализовать обработчики событий BeginPrint и EndPrint.

Событие PrintPage

Необходимо реализовать обработчик событий для события PrintPage, аргументы которого описываются делегатом PrintPageEventHandler:

```
public delegate void PrintPageEventHandler(object sender,
    PrintPageEventArgs e);
```

Мы получаем объект типа PrintPageEventArgs. На диаграмме классов можно увидеть основные свойства этого класса, у которого имеются отношения с классами PageSettings и Graphics; причем первый из них позволяет задавать размеры листа бумаги, поля и получать информацию об используемом принтере. Класс Graphics предоставляет возможность непосредственного доступа к устройству, он позволяет посылать принтеру строки, а также прямые и кривые линии.

> *GDI расшифровывается как Graphics Device Interface (интерфейс графического устройства) и предоставляет определенные возможности для вывода графической информации на такие устройства, как экран или принтер. GDI+ — это следующее поколение GDI, в котором реализованы такие возможности, как плавно изменяющие свой градиент кисти и эффекты, связанные с затемнением. Более подробную информацию о выводе графической информации с помощью GDI+ и класса Graphics можно почерпнуть из главы 16.*

Поработав со следующим примером вы сможете убедиться, что включение в приложение возможностей, связанных с выводом на печать,— очень простая задача.

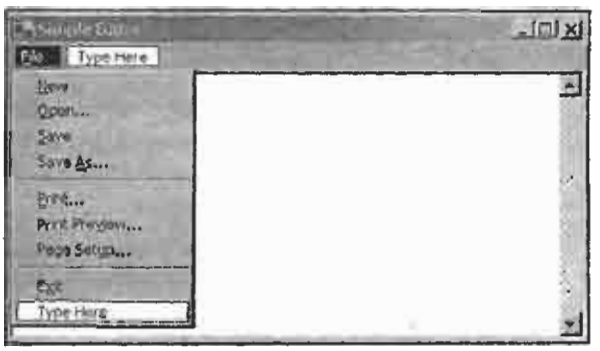
Практикум: добавление в меню вхождения Print

Название пункта меню	Текст
miFilePrint	&Print...
miFilePrintPreview	Print Pre&view...
miFilePageSetup	Page Set&up...
miFileExit	E&xit

В таблице перечислены свойства Name и Text новых пунктов меню.

На рисунке справа показано, как такое меню должно выглядеть.

Перед тем, как добавлять диалог PrintDialog, нам необходимо включить в меню некоторые вхождения, предназначенные для вывода на печать. Добавьте два разделителя и следующие пункты меню: Print, Print Preview, Page Setup и Exit.



Практикум: добавление компонента PrintDocumentи обработчика

1. Перенесите компонент PrintDocument из окна с инструментами и поместите его в форму. Замените имя на printDocument и добавьте обработчик событий OnPrintPage() в событие PrintPage, выбрав кнопку Events в окне Properties. Теперь введите код следующей реализации обработчика событий:

```
private void OnPrintPage(object sender,
            System.Drawing.Printing.PrintPageEventArgs e)
{
   char[] param = {'\n'};
   string[] lines = textBoxEdit.Text.Split(param);

   int i = 0;
   char[] trimParam = {'\r'};
   foreach (string s in lines)
   {
      lines[i++] = s.TrimEnd(trimParam);
   }

   int x = 20;
   int y = 20;
   foreach (string line in lines)
   {
      e.Graphics.DrawString(line, new Font("Arial", 10),
                            Brushes.Black, x, y);
      y += 15;
   }
}
```

2. Добавьте обработчик событий в событие Click меню Print, который будет вызывать метод Print() класса PrintDocument:

```
private void miFilePrint_Click(object sender,
   System.EventArgs e)
{
   printDocument.Print();
}
```

3. Теперь можно построить и запустить приложение, которое позволит отпечатать какой-либо документ. Для того чтобы это приложение могло работать, необходимо наличие какого-либо инсталлированного принтера.

Как это работает

Метод `Print()` объекта `printDocument` вызывает наступление события `PrintPage` с помощью класса `PrintController`:

```
printDocument.Print();
```

В обработчике `OnPrintPage()` мы последовательно разбиваем текст, находящийся в текстовом окне, на отдельные строки посредством метода `String.Split()` и символа новой строки `\n`. Получающиеся в результате этого строки заносятся в переменную `lines`:

```
private void OnPrintPage(object sender,
   System.Drawing.Printing.PrintPageEventArgs e)
{
   char[] param = ('\n');
   string[] lines = textBoxEdit.Text.Split(param);
```

В зависимости от того, каким образом создавался текстовый файл, строки могут отделяться одна от другой не только символом `\n`, но и символом `\r`. С помощью метода `TrimEnd()` символ `\r` удаляется из каждой строки:

```
int i = 0;
char[] trimParam = ('\r');
foreach (string line in lines)
{
   lines[i++] = s.TrimEnd(trimParam);
}
```

Во втором операторе `foreach` в коде, приведенном ниже, можно увидеть, что мы проходим по всем строкам и отправляем каждую из них на принтер, обращаясь к методу `e.Graphics.DrawString()`. `e` — это переменная типа `PrintPageEventArgs`, у которой свойство `Graphics` привязано к контексту принтера. Контекст принтера позволяет выводить рисунки на печатающем устройстве, а класс `Graphics` обладает определенными методами, позволяющими рисовать с помощью этого контекста.

> *Рисование и методы, используемые в классе `Graphics`, более подробно освещаются в следующей главе.*

Поскольку на настоящий момент мы еще не умеем выбирать принтер, то в данном случае будет использоваться принтер по умолчанию, информация о котором храниться в реестре Windows.

С помощью метода `DrawString()` мы задаем использование при выводе на принтер шрифта Arial размером 10 пунктов и кисть черного цвета. Координаты вывода определяются переменными `x` и `y`. Для позиции по горизонтали зафиксировано значение 20 пикселей, а позиция по вертикали увеличивается для каждой следующей строки:

```
int x = 20;
int y = 20;
foreach (string line in lines)
{
   e.Graphics.DrawString(line, new Font("Arial", 10),
                         Brushes.Black, x, y);
   y += 15;
}
```

Реализованная на данном этапе процедура вывода на печать не лишена определенных недостатков:

- Выходной поток печати направляется на принтер по умолчанию, который задается пользователем с помощью Control Panel. В приложении должен быть способ, позволяющий пользователю выбирать принтер. Для решения этой проблемы мы воспользуемся классом `PrintDialog`.
- Используемый шрифт зафиксирован. Чтобы предоставить пользователю возможность самому выбирать используемый шрифт, прибегнем к помощи класса `FontDialog`, который более подробно рассматривается ниже в настоящей главе.
- Для определения границ выводимых на печать страниц используются жестко заданные в программе значения. Есть возможность позволить пользователю выбирать другие границы страниц, для чего нужно использовать класс `PageSetupDialog`.
- Не предусмотрен вывод на печать нескольких страниц. Если выводящийся на печать документ состоит из нескольких страниц, то будет напечатана только первая из них. Нужно также предусмотреть возможность вывода на печать некоторого заголовка (например имени файла) и некоторой сноски (например номера страницы).

Продолжим работу над процедурой вывода на печать, с тем чтобы решить означенные проблемы.

Вывод на печать нескольких страниц

Событие `PrintPage` вызывается для каждой страницы, подлежащей выводу на печать. Необходимо передать `PrintController` информацию о том, что страница, выводящаяся на печать в данный момент времени, не является последней, посредством присваивания свойству `HasMorePages` класса `PrintPageEventArgs` значения `true`.

Практикум: модификация OnPrintPage для вывода нескольких страниц

1. Необходимо объявить переменную `lines` типа `string[]` и переменную `linesPrinted` типа `int` в классе `SimpleEditorForm`:

```
private string[]lines;
private int linesPrinted;
```

2. Внесите следующие изменения в обработчик `OnPrintPage()`. В реализации `OnPrintPage()`, которая использовалась до настоящего момента, мы осуществляли разбиение текста на строки. Поскольку метод `OnPrintPage()` вызывается для каждой страницы, разбиение текста на строки оказывается необходимым выполнять только один раз — перед началом процедуры вывода на печать. Удалите весь код из метода `OnPrintPage()` и вместо него введите новую реализацию:

```
private void OnPrintPage(object sender,
                         System.Drawing.Printing.PrintPageEventArgs e)
{
    int x = 20;
    int y = 20;
```

```
        while (linesPrinted < lines.Length)
        {
            e.Graphics.DrawString(lines[linesPrinted++],
                        new Font("Arial", 10), Brushes.Black, x, y);
            y += 15;
            if (y >= e.PageBounds.Height - 80)
            {
                e.HasMorePages = true;
                return;
            }
        }

        linesPrinted = 0;
        e.HasMorePages = false;
    }
```

3. Добавьте обработчик события BeginPrint объекта printDocument. Метод OnBeginPrint() вызывается всего один раз для каждого задания вывода на печать, и в нем мы создаем массив lines:

```
private void OnBeginPrint(object sender,
                          System.Drawing.Printing.PrintEventArgs e)
{
    char[] param = {'\n'};
    lines = textBoxEdit.Text.Split(param);

    int i = 0;
    char[] trimParam = {'\r'};
    foreach (string s in lines)
    {
        lines[i++] = s.TrimEnd(trimParam);
    }
}
```

4. Построив проект, вы можете приступать к осуществлению задачи вывода на печать многостраничного документа.

Как это работает

Выполнение задачи вывода на печать начинается с вызова метода Print() класса PrintDocument, который, в свою очередь, сначала единственный раз вызывает метод BeginPrint(), а затем — метод OnPrintPage() для каждой страницы.

В методе BeginPrint() производится разбиение текста, находящегося в текстовом окне, на массив строк. Каждая строка в массиве представляет собой отдельную строку, поскольку мы выполняем разбиение по символу перехода к новой строке (\n), а все символы возврат каретки (\r) удаляются — так же, как это делалось раньше.

```
char[] param = {'\n'};
lines = textBoxEdit.Text.Split(param);

int i = 0;
char[] trimParam = {'\r'};
foreach (string s in lines)
{
    lines[i++] = s.TrimEnd(trimParam);
}
```

Метод OnPrintPage() вызывается после метода BeginPrint(). Нам требуется продолжать вывод на печать, пока число отпечатанных строк меньше числа строк, выводимых на печать. Свойство lines.Length позволяет получить число строк,

находящихся в массиве lines. Переменная linesPrinted увеличивается на единицу с каждой отправленной на печать строкой:

```
while (linesPrinted < lines.Length)
{
   e.Graphics.DrawString(lines[linesPrinted++],
                         new Font("Arial", 10), Brushes.Black, x, y);
```

После того, как строка отпечатана, мы осуществляем проверку, не выходит ли вновь вычисленное значение позиции по вертикали за границы страницы. Кроме того, мы уменьшаем границы на 80 пикселей, поскольку не хотим, чтобы печать производилась по самому нижнему краю страницы, так как многие принтеры за этим не следят. Если такая позиция достигнута, то свойству HasMorePages класса PrintPageEventArgs присваивается значение true для того, чтобы проинформировать контроллер о том, что метод OnPrintPage() должен быть вызван еще раз. Не забудьте, что в методе PrintController используется метод PrintLoop(), который определяет последовательность выводимой информации для каждой печатающейся страницы. Если значение свойства HasMorePages будет равно false, он прекратит работу. В качестве значения свойства HasMorePages по умолчанию используется false, чтобы печать была прекращена:

```
y += 15;
if (y >= e.PageBounds.Height)
{
   e.HasMorePages = true;
   return;
}
```

Задание параметров страницы

На данном этапе значения, определяющие поля страницы, пр чно "зашиты" в программу. Давайте ее модифицируем так, чтобы пользователь сам определял поля страницы. Для этого воспользуемся PageSetupDialog — еще одним классом диалога.

Этот класс позволяет задавать размеры и источник поступления бумаги, ориентацию листов, поля страницы. Поскольку все эти опции зависят от конкретного принтера, с помощью этого диалога можно также осуществлять и выбор самого принтера.

Бумага

Если свойству AllowPaper присвоено значение true, то это означает, что пользователь имеет возможность самостоятельно определять размеры листов бумаги и ее источник. Свойство PageSetupDialog.PageSettings.PaperSize возвращает экземпляр PaperSize, с котором хранятся высота, ширина и тип бумаги соответственно в свойствах Height, Width и PaperName. PaperName определяется именами Letter и A4. Свойство Kind имеет значение перечислимого типа PaperKind. Оно может принимать одно из трех следующих значений, определяющих размеры используемой бумаги: European (европейские), American (американские) и Japanese (японские).

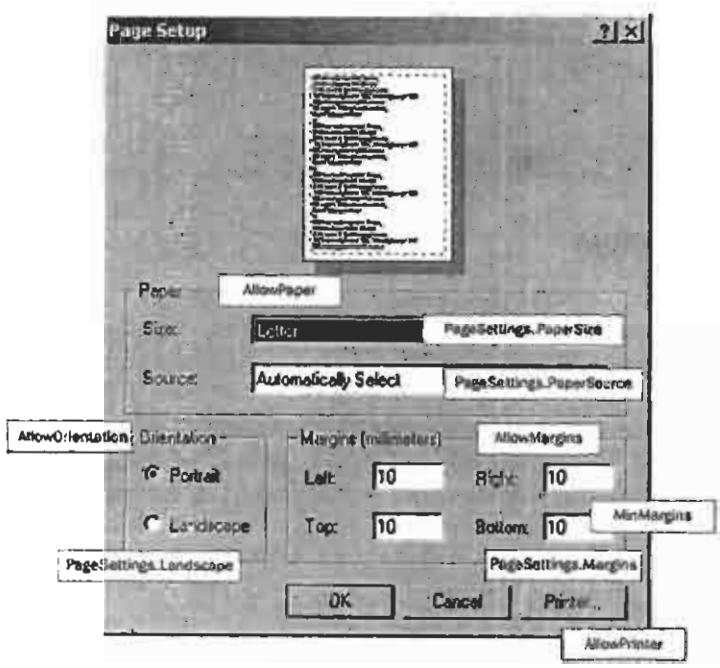
Свойство PageSetupDialog.PageSettings.PaperSource возвращает экземпляр PaperSource, в котором находится название источника бумаги для принтера, и того, какая именно бумага там находится (при условии, что установки принтера сконфигурированы корректно).

Поля

Если свойству `AllowMargins` присвоено значение `true`, то это означает, что пользователь имеет возможность самостоятельно определять размеры полей при выводе на печать. С помощью свойства `MinMargins` существует возможность установить минимально допустимые значения, которые могут быть введены пользователем. Для считывания значений, определяющих размеры полей, используется свойство `PageSetupDialog.PageSettings.Margins`. Возвращаемый объект `Margins` обладает свойствами `Bottom`, `Left`, `Right` и `Top`.

Ориентация листов

Свойство `AllowOrientation` определяет, имеется ли у пользователя возможность выбирать между горизонтальной и вертикальной ориентацией листа бумаги при выводе на печать. Выбранное значение может быть получено с помощью логического свойства `PageSetupDialog.PageSettings.Landscape`, значение `true` которого соответствует горизонтальному расположению листа бумаги, а `false` — вертикальному.



Принтер

Свойство `AllowPrinter` определяет, имеет ли пользователь возможность выбирать принтер. В зависимости от значения этого свойства разрешается или запрещается использование кнопки **Printer**. Обработчик этой кнопки, в свою очередь, открывает диалоговое окно **PrintDialog**, которое мы будем рассматривать следующим.

Свойства PageSetupDialog

На рисунке слева представлен обзор свойств, которые позволяют или запрещают использование различных возможностей, а также указано, какие свойства могут быть использованы для получения соответствующих значений.

Практикум: добавление диалогового окна для задания свойств страницы

1. Перенесите компонент `PageSetupDialog` из окна с инструментами в программу разработки Windows Forms. Присвойте свойству `Name` значение `dlgPageSetup`, а свойству `Document` — `printDocument`, для того чтобы привязать к диалогу документ, подлежащий выводу на печать.

   Затем добавьте обработчик событий "щелчок мыши" во вхождение меню **Page Setup**, а также код, ответственный за вывод диалогового окна на экран с помощью метода `ShowDialog()`. В данном случае необходимость проверять возвращаемые методом `ShowDialog()`

значения отсутствует, поскольку используемая реализация обработчика событий "нажатие кнопки OK" уже предполагает присваивание новых значений привязанному объекту `printDocument`:

```
private void menuItemFilePageSetup_Click(object sender,
                                         System.EventArgs e)
{
   dlgPageSetup.ShowDialog();
}
```

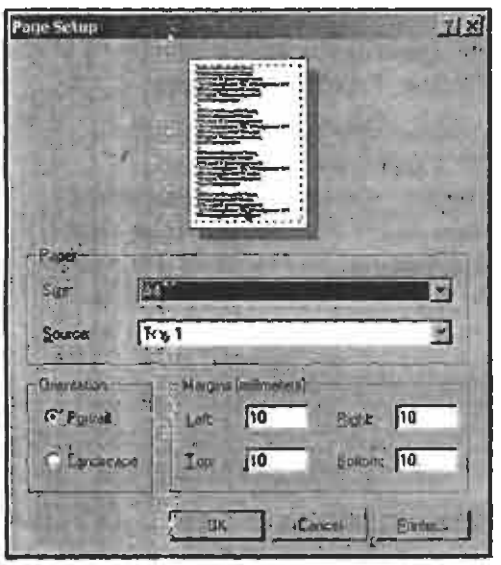
2. Теперь необходимо внести изменения в реализацию метода `OnPrintPage()`, которые позволят использовать значения размеров полей, заданные с помощью диалога `PageSetupDialog`. В используемой нами программе значения переменных `x` и `y` присваиваются свойствам `MarginBounds.Left` и `MarginBounds.Right` класса `PrintPageEventArgs`. Проверка на предмет достижения конца страницы осуществляется с помощью свойства `MarginBounds.Bottom`:

```
private void OnPrintPage(object sender,
                  System.Drawing.Printing.PrintPageEventArgs e)
{
   int x = e.MarginBounds.Left;
   int y = e.MarginBounds.Top;

   while (linesPrinted < lines.Length)
   {
      e.Graphics.DrawString(lines[linesPrinted++],
                     new Font("Arial", 10), Brushes.Black, x, y);

      y += 15;
      if (y >= e.MarginBounds.Bottom)
      {
         e.HasMorePages = true;
         return;
      }
   }

   linesPrinted = 0;
   e.HasMorePages = false;
}
```



3. Теперь можно построить проект и запустить приложение. Выбрав **File | Page Setup**, вы получите диалог, показанный ниже. Если вы измените границы, то печать будет осуществляться в этих измененных границах (см. рис. слева).

Если вывод диалогового окна PageSetupDialog на экран оказывается невозможным, то генерируется исключительная ситуация типа `System.ArgumentException`; это, в частности, может происходить потому, что вы забыли привязать объект `PrintDocument` к диалогу `PageSetupDialog`. Диалогу `PageSetupDialog` наличие привязанного объекта `PrintDocument` необходимо — он использует его для считывания и записи различных значений, используемых в процессе диалога.

Диалог вывода на печать

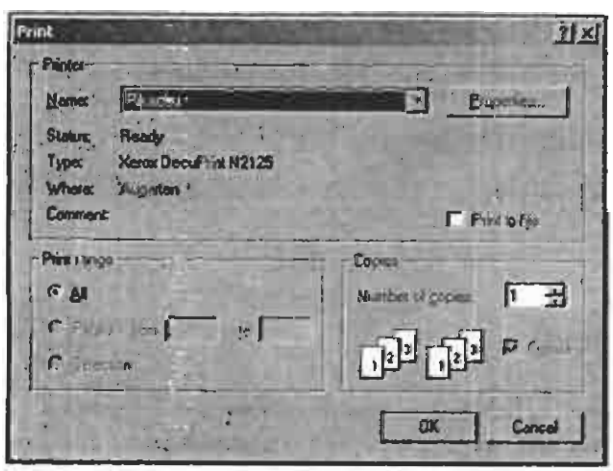
Класс `PrintDialog` позволяет пользователю выбирать принтер из числа инсталлированных, а также задавать число копий и некоторые установки принтера, например представление вывода и источник бумаги для принтера. Поскольку использовать диалог `PrintDialog` очень просто, мы сразу переходим к процедуре включения этого диалога в наше приложение `Editor`.

Практикум: добавление диалогового окна PrintDialog

1. Добавьте в форму компонент `PrintDialog` из окна с инструментами. Измените его свойство `Name` на `dlgPrint`, а свойство `Document` — на `printDocument`.

   Замените реализацию обработчика событий "щелчок мышью" для меню Print на следующий код:

```
private void miFilePrint_Click(object sender, System.EventArgs e)
{
   if (dlgPrint.ShowDialog() == DialogResult.OK)
   {
      printDocument.Print();
   }
}
```



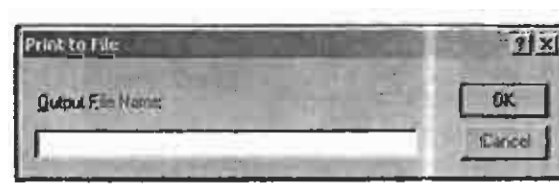
2. Постройте и запустите приложение. Выбрав File | Print, вы откроете диалог `PrintDialog`. Теперь есть возможность выбрать принтер, на котором вы будете распечатывать документ (см. рис. слева).

Опции диалогового окна вывода на печать

В программе `SimpleEditor` мы не вносили никаких изменений в свойства `PrintDialog`. Но у этого диалогового окна также имеется несколько опций. В приведенном выше диалоговом окне имеющиеся опции разбиты на три группы: Printer, Print Range и Copies.

- В группе Printer можно выбрать не только принтер — в ней также существует опция Print to File (вывод в файл). По умолчанию эта опция находится в рабочем состоянии, но не является выбранной. Установка флажка в соответствующем окне выбора дает возможность пользователю направлять выходной поток, предназначенный для вывода на печать, в некоторый файл вместо принтера. Эта опция может быть отключена, если присвоить свойству `AllowPrintToFile` значение `false`.

Если пользователь выбирает эту опцию, то метод `printDocument.Print()` открывает диалоговое окно, представленное на рисунке справа. В названном окне запрашивается имя файла, в который следует записывать выходной поток, предназначенный для вывода на печать.



- В разделе диалогового окна Print Range можно выбрать только опцию All (все) — опции Pages и Selection по умолчанию отключены. С тем, каким образом эти опции могут быть реализованы, мы познакомимся в следующем разделе.
- Группа Copies позволяет задавать число получаемых копий.

Вывод на печать выбранного текста

Если требуется реализовать возможность вывода на печать выбранного только текста, то для этого необходимо активизировать соответствующий переключатель, присвоив свойству `AllowSelection` значение `true`. Для этого также необходимо изменить код, отвечающий за вывод на печать, с тем чтобы он выводил на печать только выбранный текст.

Практикум: добавление возможности вывода на печать выбранного текста

1. Добавьте код, выделенный серым цветом, в обработчик событий "щелчок мышью" кнопки Print:

```
private void miFilePrint_Click(object sender, System.EventArgs e)
{
   if (textBoxEdit.SelectedText != "")
   {
      dlgPrint.AllowSelection = true;
   }
   if (dlgPrint.ShowDialog() == DialogResult.OK)
   {
      printDocument.Print();
   }
}
```

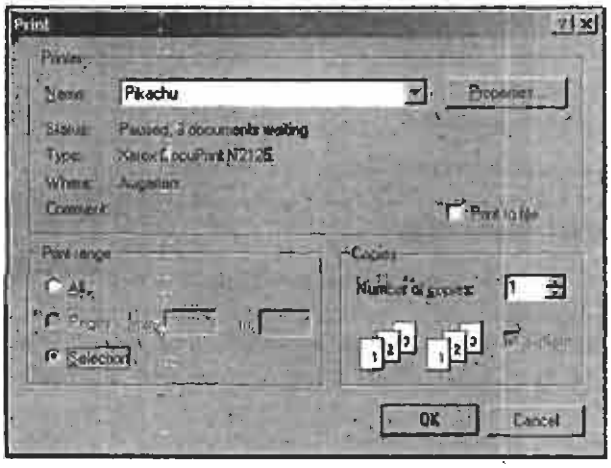
В нашей программе все строки, подлежащие выводу на печать, задаются в обработчике `OnBeginPrint()`. В реализацию этого метода также необходимо внести некоторые изменения:

```
private void OnBeginPrint(object sender,
                          System.Drawing.Printing.PrintEventArgs e)
{
   char[] param = {'\n'};

   if (dlgPrint.PrinterSettings.PrintRange == PrintRange.Selection)
   {
      lines = textBoxEdit.SelectedText.Split(param);
   }
   else
   {
      lines = textBoxEdit.Text.Split(param);
   }

   int i = 0;
   char[] trimParam = {'\r'};
   foreach (string s in lines)
   {
      lines[i++] = s.TrimEnd(trimParam);
   }
}
```

Теперь можно построить и запустить программу.
Откройте какой-либо файл, выберите некоторый текст, запустите диалоговое окно вывода на печать с помощью меню File | Print



и выберите кнопку Selection в группе Print Range (см. рис. слева). Если с такими установками нажать на кнопку Print, то на печать будет выведен только выбранный текст.

Как это работает

Мы присваиваем свойству `AllowSelection` значение `true` только в том случае, если есть какой-либо выбранный текст. Перед тем как открыть `PrintDialog`, необходимо проверить, выбран какой-либо текст или нет, т. е. убедиться в том, что значение свойства `SelectedText` не есть `null`. Если есть какой-либо выбранный текст, то свойству `AllowSelection` присваивается значение `true`:

```
if (textBoxEdit.SelectedText != "")
{
   dlgPrint.AllowSelection = true;
}
```

Метод `OnBeginPrint()` вызывается в начале каждого задания по выводу на печать. Обращаясь к свойству `PrintDialog.PrinterSettings.PrintRange` мы получаем информацию о том, что пользователем выбрана опция Selection. Свойство `PrintRange` возвращает значение перечислимого типа `PrintRange`, которое может быть одним из следующих: `AllPages` (все страницы), `Selection` (выбор) или `SomePages` (некоторые страницы):

```
if (print.Dialog.PrinterSettings.PrintRange == PrintRange.Selection)
{
```

Если выбрана опция Selection, мы берем строку из выбранного текста, обладающую свойством `SelectedText` класса `Textbox`. Эта строка подвергается точно такому же разбиению, как и весь текст:

```
   lines = textBoxEdit.SelectedText.Split(param);
}
```

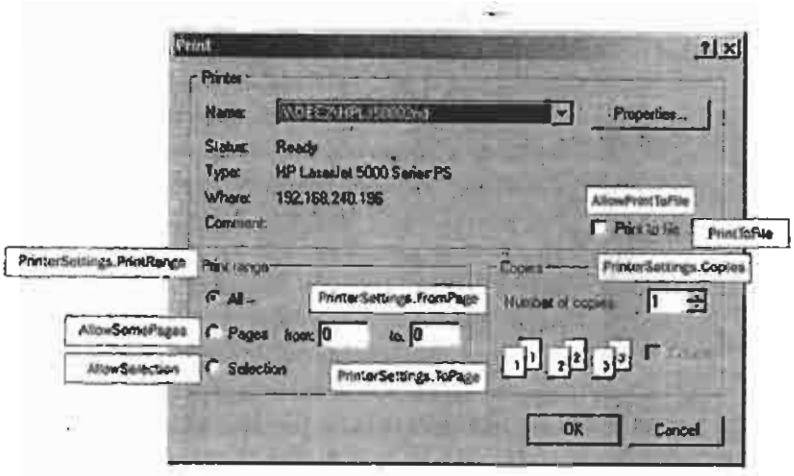
Для того, чтобы иметь возможность использовать перечислимый тип `PrintRange`, необходимо включить пространство имен `System.Drawing.Painting` в самое начало программы:

```
using System.Drawing.Painting;
```

Вывод на печать диапазонов страниц

Вывод на печать определенного диапазона страниц может быть реализован способом, аналогичным выводу на печать выбранного текста. Кнопка выбора данной опции может быть активизирована путем присваивания свойству `AllowSomePages` значения `true`. В этом случае пользователь получает возможность выбрать диапазон страниц, подлежащих выводу на печать. Каким образом можно определить границы страниц в программе Simple Editor? Какая страница является последней? Для того чтобы задать последнюю страницу, необходимо присвоить соответствующее значение свойству `PrintDialog.PrinterSettings.ToPage`. Каким образом пользователь сможет узнать номера страниц, которые ему требуется вывести на печать?

Это не составляет никакого труда в приложениях, предназначенных для работы с документами, например в Microsoft Word, в котором в качестве режима вывода на экран может быть выбран Print Layout. Простой управляющий элемент `Textbox`, используемый в нашем приложении Simple Editor, не предоставляет таких возможностей. Именно по этой причине мы не будем реализовывать данную возможность в нашем приложении.



Свойства PrintDialog

Сведем все свойства, оказывающие влияние на представление `PrintDialog`, на одном рисунке (см. рис. слева).

Предварительный просмотр

Возможно, пользователь пожелает посмотреть, как будет выглядеть выводимая на печать информация. Именно для этой цели используется Print Preview (предварительный просмотр). Реализовать Print Preview в .NET оказывается очень просто. Для этого можно воспользоваться классом `PrintPreviewControl`, который позволяет осуществлять предварительный просмотр документа в том виде, в котором он будет выводиться на печать. `PrintPreviewControl` — это диалог, который представляет собой оболочку для данной возможности.

Диалоговое окно PrintPreviewDialog

Взглянув на свойства и список наследования класса `PrintPreviewDialog` в документации MSDN, вы легко сможете обнаружить, что они представляют собой самостоятельную форму, а не оболочку для общего диалога — этот класс является производным от `System.Windows.Forms.Form`, и работать с ним следует так же, как с формами, созданными в предыдущей главе.

Мы включим класс `PrintPreviewDialog` в приложение Simple Editor.

Практикум: добавление диалогового окна предварительного просмотра

1. Добавьте в программу разработки Windows Forms компонент `PrintPreviewDialog`. Присвойте ему имя `dlgPrintPreview`, а свойству `Document` — значение `printDocument`. Добавьте и реализуйте обработчик события `Click` для вхождения меню Print Preview (см. рис. справа):

```
private void miFilePrintPreview_Click( object
                    sender, System.EventArgs e)
{
   dlgPrintPreview.ShowDialog();
}
```



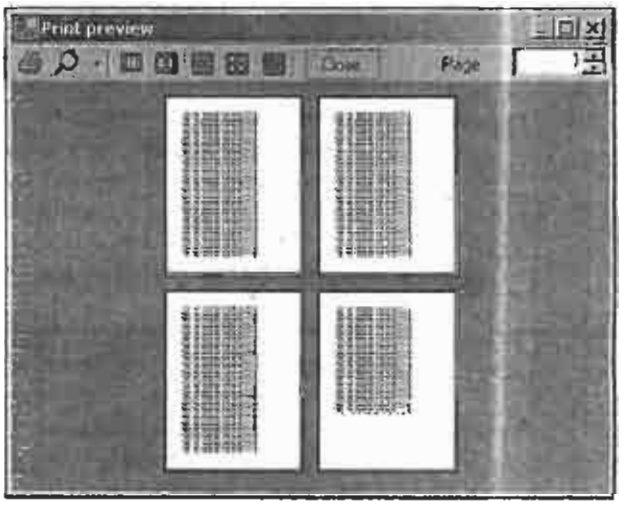
Класс PrintPreviewControl

Предварительный просмотр в приложениях Microsoft Word и WordPad отличается от `PrintPreviewDialog` тем, что в этих приложениях для него не используется отдельное диалоговое окно — просмотр осуществляется в основном окие этих приложений.



Чтобы добиться такого эффекта, можно поместить в форму класс `PrintPreviewControl`. Свойству `Document` следует присвоить объект `PrintDocument`, а свойству `Visible` — значение `false`. В обработчике можно присвоить свойству `Visible` значение `true` для предварительного просмотра документа. В этом случае средство управления `PrintPreviewControl` заслонит собой все остальные средства управления, как показано на рисунке слева.

По названию и единственному пункту меню File можно определить, что это — основное окно приложения Simple Editor. Теперь осталось добавить еще несколько элементов, которые в классе `PrintPreviewControl` отвечают за увеличение изображения, печать и вывод 2, 3, 4, 6 и т. д. страниц. Для того чтобы воспользоваться этими возможностями следует обратиться к особой панели инструментов. В классе `PrintPreviewDialog` это все уже реализовано, как это видно на рисунке справа, на котором изображен предварительный просмотр четырех страниц (см. рис. справа).



Диалоговые окна FontDialog и ColorDialog

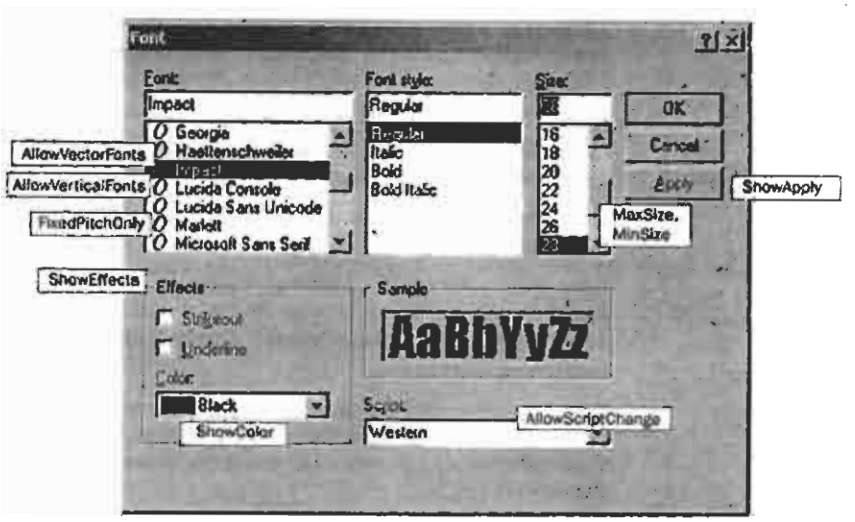
Последние два диалоговых окна, на которых мы коротко остановимся в данной главе,— это `FontDialog` и `ColorDialog`.

> *Основное внимание будет сосредоточено на обсуждении диалоговых окон, позволяющих задавать шрифт и цвет, а не на классах `Font` и `Color`, поскольку эти классы детально изучаются в главе 16.*

Диалоговое окно FontDialog

Диалоговое окно `FontDialog` позволяет пользователю приложения осуществлять выбор шрифта. Пользователю предоставляется возможность изменять шрифт, стиль, размер и цвет шрифта.

На следующем рисунке показаны все свойства, которые отвечают за изменение элементов данного диалогового окна:



Использование диалогового окна FontDialog

Данное диалоговое окно может использоваться точно так же, как и все предыдущие. В программе разработки Windows Forms это окно можно перенести из окна с инструментами и поместить в форму таким образом, что это приведет к созданию экземпляра `FontDialog`.

Код, предназначенный для использования `FontDialog`, может выглядеть следующим образом:

```
if (dlgFont.ShowDialog() == DialogResult.OK)
{
   textBoxEdit.Font = dlgFont.Font;
}
```

Диалоговое окно FontDialog выводится с помощью метода `ShowDialog()`. Если пользователь завершает диалог нажатием кнопки OK, то этим методом возвращается значение `DialogResult.OK`. Выбранный пользователем шрифт можно определить, воспользовавшись свойством `Font` класса `FontDialog`; в дальнейшем этот шрифт передается свойству `Font` класса `Textbox`.

Свойства диалогового окна FontDialog

Назначение свойств класса `FontDialog` можно узнать из следующей таблицы.

Свойство	Описание
`AllowVectorFonts`	Логическое значение, определяющее, могут ли из списка шрифтов выбираться векторные шрифты.
`AllowVerticalFonts`	Логическое значение, определяющее, могут ли из списка шрифтов выбираться вертикальные шрифты. Вертикальная запись текстов используется в некоторых дальневосточных странах. Вполне вероятно, что на вашем компьютере нет ни одного инсталлированного вертикального шрифта.

Продолжение таблицы

Свойство	Описание
FixedPitchOnly	Установка этого свойства приведет к тому, что в списке шрифтов будут выводиться только шрифты с фиксированными размерами. У шрифтов с фиксированными размерами все буквы обладают одинаковыми размерами. Значение по умолчанию — false.
MaxSize	Позволяет определить максимальный размер шрифта, который может выбрать пользователь.
MinSize	Позволяет определить минимальный размер шрифта, который может выбирать пользователь.
ShowApply	Если требуется вывести кнопку Apply, то данному свойству следует присвоить значение true. Нажав кнопку Apply, пользователь получает возможность посмотреть, как выглядит новый шрифт в приложении, не покидая диалога выбора шрифтов.
ShowColor	По умолчанию возможность выбора цвета в данном диалоге отсутствует. Если вы хотите предоставить пользователю возможность выбирать цвет в диалоге выбора шрифта, то свойству ShowColor должны быть присвоено значение true.
ShowEffects	По умолчанию можно устанавливать флажки в окошках выбора элементов Strikeout (зачеркивание) и Underline (подчеркивание) для выполнения манипуляций над шрифтами. Если вы не желаете, чтобы эти возможности выводились на экран, то свойству ShowEffects следует присвоить значение false.
AllowScriptChange	Присваивание свойству AllowScriptChange значения false налагает запрет на изменение пользователем стиля выбранного шрифта. Стили, которые доступны пользователю, зависят от выбранного им шрифта. Например, шрифт Arial поддерживает западный, еврейский, арабский, греческий, турецкий, балтийский, центрально-европейский, кириллический и вьетнамский шрифты.

Разрешение использования кнопки Apply

Интересной отличительной особенностью диалогового окна FontDialog от других рассматривавшихся окон является кнопка Apply, которая по умолчанию не выводится. Если пользователь нажимает кнопку Apply, то диалоговое остается открытым, однако используется выбранный шрифт.

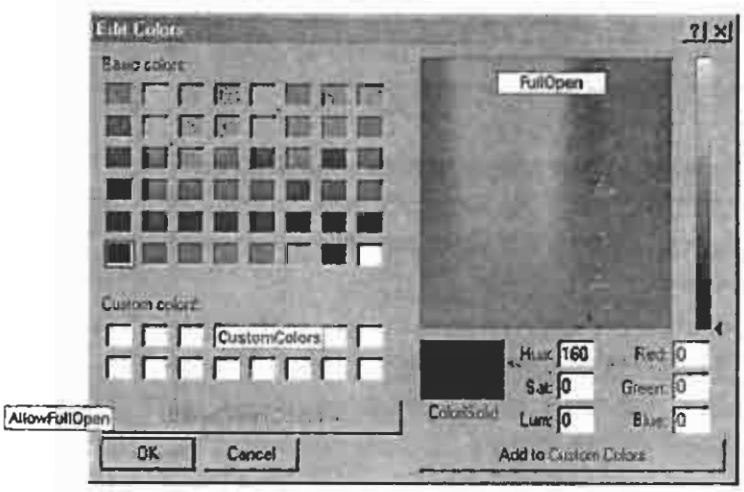
Выбрав FontDialog в программе разработки Windows Forms, можно присвоить свойству ShowApply значение true с помощью окна Properties. Каким образом можно узнать, что пользователь нажал кнопку Apply? Диалоговое окно по-прежнему остается открытым, и метод ShowDialog() не возвращает никаких значений. Вместо этого можно добавить обработчик событий для события Apply класса FontDialog, нажав кнопку Events в окне Properties и введя имя обработчика в событие Apply.

Как станет ясно из приведенного далее кода, автор использовал имя OnApplyFontDialog. Этот обработчик позволяет получить информацию о том, какой именно шрифт выбран в диалоговом окне FontDialog с помощью переменной класса FontDialog:

```
private void OnApplyFontDialog(object sender, System.EventArgs e)
{
    textBoxEdit.Font = dlgFont.Font;
}
```

Диалоговое окно ColorDialog

В диалоговом окне ColorDialog не так много настраиваемых возможностей, как в FontDialog. ColorDialog позволяет пользователю создать свои собственные цвета, если его не устраивает ни один из предлагаемых базовых цветов; это достигается установкой свойства `AllowFullColor`. Часть диалогового окна, отвечающая за создание цветов, может быть автоматически расширена посредством свойства `FullColor`. Свойство `SolidColorOnly` указывает на то, что пользователем могут выбираться только однородные цвета. Свойство `CustomColors` может быть использовано для считывания и записи новых значений параметров создаваемого цвета (см. рис. слева).



Использование диалогового окна выбора цветов

Диалоговое окно ColorDialog может быть перенесено мышью из окна с инструментами и помещено в программу разработки Windows Forms точно так же, как и другие. Метод `ShowDialog()` будет выводить этот диалог на экран до тех пор, пока не будет нажата кнопка OK или кнопка Cancel. Узнать, какой цвет выбран пользователем, можно, считав значение свойства `Color` данного диалога, как это продемонстрировано в следующем примере:

```
if (dlgColor.ShowDialog() == DialogResult.OK)
{
   textBoxEdit.ForeColor = dlgFont.Color;
}
```

Свойства диалогового окна ColorDialog

Все свойства, которые оказывают влияние на внешнее представление этого окна, сведены в следующую таблицу:

Свойства	Описание
`AllowFullOpen`	Присвоив этому свойству значение `false`, можно запретить пользователю самостоятельно определять цвета, отключив кнопку Define Custom Colors. Значение этого свойства по умолчанию — `true`.
`FullOpen`	Присвоив этому свойству значение `true`, перед тем как диалог будет выведен, можно добиться автоматического открытия диалогового окна с возможностью создания собственных цветов.
`AnyColor`	Присвоив этому свойству значение `true`, можно получить все допустимые цвета в списке основных цветов.
`CustomColors`	С помощью этого свойства можно создавать массивы заранее определенных цветов и считывать цвета, определенные пользователем.
`SolidColorOnly`	Если этому свойству присвоено значение `true`, то пользователь имеет возможность выбирать только однородные цвета.

Итоги

В настоящей главе мы познакомились с тем, каким образом в приложениях можно использовать классы диалоговых окон. Мы узнали, как можно открывать и сохранять файлы, сделали обзор классов, которые используются в .NET Framework для вывода на печать, а затем рассказали о том, каким образом можно включать в приложения различные возможности для вывода на печать. Подводя итоги, перечислим те классы диалоговых окон, которые мы использовали в нашем приложении Simple Editor:

- `OpenFileDialog` используется для ввода пользователем имени открываемого файла.
- `SaveFileDialog` используется для ввода пользователем имени сохраняемого файла
- `PrintDialog` используется для выбора принтера, на котором будет осуществляться печать и для задания конфигурации печати
- `PageSetupDialog` используется для изменения параметров полей страницы, на которой осуществляется печать
- `PrintPreviewDialog` используется для предварительного просмотра выводимой на печать информации, что позволяет пользователю заранее знать, каким образом будет выглядеть отпечатанный документ
- Мы также познакомились с основными возможностями классов `FontDialog` и `ColorDialog`. Включение этих классов в приложение Simple Editor будет использовано в качестве одного из упражнений.

Упражнения

Поскольку диалоговые окна FontDialog и ColorDialog работают точно так же, как и другие, рассматривавшиеся в этой главе, задача включения их в приложение Simple Editor не представляет никаких сложностей:

1. Предусмотрите возможность для пользователя изменять шрифт текстового окна. Для реализации такой возможности введите новое вхождение основного меню — F&ormat, и субменю Format — &Font... . Добавьте в этот пункт меню обработчик событий. Включите в приложение диалоговое окно FontDialog с помощью программы разработки Windows Forms. Выведите это окно с помощью обработчика событий меню и присвойте выбранный шрифт в качестве значения свойству `Font` текстового окна.

   Вам также придется внести изменения в реализацию метода `OnPrintPage()`, которые позволят использовать выбранный шрифт при выводе на печать. В предшествующей реализации нами был создан новый объект `Font` в методе `DrawString()` объекта `Graphics`. В данном случае мы вместо этого используем шрифт объекта `textBoxEdit`, считывая значение свойства `Font`. Также необходимо учитывать возможность возникновения проблем, если пользователь выбирает слишком большой шрифт. Чтобы избежать ситуацию, при которой текущая строка будет частично налагаться на другую строку,

расположенную ниже или выше, следует внести соответствующие изменения в фиксированное значение, которое использовалось для определения позиции строк по вертикали. Наилучшим способом будет использование в качестве величины, на которую увеличивается вертикальная координата строки, значение размера шрифта: используйте свойство `Height` класса `Font`.

2. Другим существенным расширением приложения Simple Editor будет включение в него возможности изменять цвет шрифта. Добавьте второе подменю во вхождение меню Format — Color... . Добавьте обработчик для данного вхождения меню, в котором осуществляется открытие диалогового окна ColorDialog. Если пользователь нажимает кнопку OK, присвойте выбранный в окне ColorDialog цвет свойству `ForeColor` текстового окна.

   В методе `OnPrintPage()` убедитесь в том, что выбранный цвет используется только в том случае, если принтер поддерживает цветную печать. Проверить, поддерживает ли принтер цветную печать, можно с помощью свойства `PageSettings.Color` аргумента `PrintPageEventArgs`. Можно создать объект типа кисти того цвета, который используется в текстовом окне, с помощью следующего кода:

```
Brush brush = new SolidBrush(textBoxEdit.ForeColor);
```

   Эта кисть может использоваться в качестве аргумента метода `DrawString()` вместо черной кисти, применявшейся в предыдущем примере.

16

Введение в GDI+

Настоящая глава является введением в программирование с использованием классов Graphics Device Interface (GDI+ — интерфейс графических устройств). Приложения с графикой, игры, Computer Aided Design/Computer Aided Manufacture (CAD/CAM — проектирование/производство с помощью компьютера), программы для рисования, для создания графиков и многие другие типы приложений требуют от разработчиков написания кода для работы с графикой. Использование создаваемых пользователем управляющих элементов также предполагает работу с графикой. Посредством своей последней библиотеки классов компания Microsoft сделала написание кода для работы с графикой как никогда простым.

Писать код для работы с графикой — это наибольшее удовольствие из всех программистских задач. Это очень здорово, когда ты вносишь изменения в программу и тут же наглядно видишь результаты этих изменений. Если вы создаете собственное графическое окно, в котором нечто представляется новым, оригинальным способом, или, если вы создаете собственный управляющий элемент, который облегчит использование вашего приложения и сделает его более приятным, то это будет благосклонно приниматься широкой публикой.

Сначала объясним механизм рисования с помощью GDI+ и напишем несколько простых примеров графических программ. А затем рассмотрим с точки зрения высокого уровня некоторые дополнительные возможности GDI+, включая:

- Работу с отдельными частями рисунков
- Вывод на печать
- Предварительный просмотр
- Пространство имен `Drawing2D`
- Пространство имен `Imaging`

После этого мы познакомимся с классами, которые необходимо использовать для реализации этих возможностей, а также укажем литературу для дальнейшего изучения. Знание возможностей и понимание иерархии классов — это уже половина успеха.

Обзор вывода графической информации

Первое, что необходимо усвоить при написании графического кода, это то, что Microsoft Windows не запоминает, каким образом выглядит данное графическое окно, если оно заслоняется другими окнами. Если закрытое ранее окно оказывается

на переднем плане и становится видимым, то система Windows сообщает нашему приложению: "Ваше окно (или некоторая его часть) становится видимым. Не будете ли так любезны нарисовать его?". На приложение возлагается только задача рисования содержимого окна. Вывод границ окна, линейки с заголовком и прочих графических составляющих самого окна система Windows возьмет на себя.

С точки зрения программирования когда мы создаем окно, в котором собираемся что-либо нарисовать, то мы, скорее всего, объявляем класс, который является производным от `System.Windows.Forms.Form`. Если мы создаем собственный управляющий элемент, мы должны объявить класс, который является производным от `System.Windows.Form.UserControl`. В обоих случаях придется переопределять виртуальную функцию `OnPaint()`. В Windows будет происходить обращение к этой функции всякий раз, когда возникает необходимость повторного рисования любой части окна.

При наступлении этого события класс `PaintEventArgs` передается в качестве аргумента. Представляют интерес два элемента информации, содержащейся в `PaintEventArgs`,— объекты `Graphics` и `ClipRectangle` (вырезанный прямоугольник). Класс `Graphics` — это следующая тема, которую мы будем изучать. Вопроса вырезания частей рисунка мы коснемся в общих чертах ближе к концу главы.

Класс Graphics

В классе `Graphics` инкапсулированы поверхности рисования GDI+. Есть три основных типа поверхностей рисования:

- Окна и управляющие элементы на экране
- Страницы, посылаемые на принтер
- Растровые изображения в памяти

В классе `Graphics` предусмотрены функции, которые позволяют рисовать на любой из этих поверхностей. Этот класс позволяет также рисовать дуги, кривые, кривые Безье (Bezier), эллипсы, рисунки, прямые, прямоугольники и текст.

Объект `Graphics` для некоторого окна можно получить двумя путями. Первый заключается в переопределении события `OnPaint()` — виртуального метода, который класс `Forms` наследует от класса `Control`. В этом случае объект `Graphics` мы получаем из `PaintEventArgs`, который передается вместе с событием:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;

   // Рисуем здесь

}
```

В некоторых ситуациях может потребоваться выполнять рисование в окне непосредственно, не дожидаясь наступления события `OnPaint()`. Это может оказаться актуальным в том случае, если мы создаем код, предназначенный для выбора в окне каких-либо графических объектов (аналогичный выбору иконок в Windows Explorer), либо переносим некий объект с помощью мыши. В этом случае доступ к объекту `Graphics` можно получить, обращаясь к методу `CreateGraphics()` данной формы, который представляет собой еще один метод, наследуемый классом `Forms` от класса `Control`:

```
protected void Form1_Click(object sender, System.EventArgs e)
{
   Graphics g = this.CreateGraphics();

   // Рисуем здесь

   g.Dispose();       // Это важный момент
}
```

Создание приложения, позволяющего переносить мышью и размешать графические объекты, оказывается весьма непростым предприятием, рассмотрение которого выходит за рамки настоящей главы. Использование названного способа встречается значительно реже. На начальных этапах мы будем почти всегда выполнять рисование в качестве реакции на наступление события `OnPaint()`.

Существуют также и другие способы получения объекта `Graphics`, но к их рассмотрению мы вернемся несколько позже.

Удаление объектов

Всегда вызывайте метод `Dispose()` для графических объектов.

Каждому из нас знакомо поведение Windows в тех случаях, когда ресурсы оказываются исчерпанными. Система начинает работать очень медленно, при этом приложения зачастую рисуются неправильно. Грамотно организованные приложения должны освобождать используемые ими ресурсы сразу же, как только они становятся ненужными. При разработке с использованием .NET Framework существует несколько типов данных, для которых обращение к методу `Dispose()` оказывается очень важным, поскольку в противном случае определенные ресурсы никогда не будут освобождены. Эти классы реализуют интерфейс `IDisposable`. Класс `Graphics` является одним из таких классов. Вызов метода `Dispose()` оказывается важным, однако, если мы получаем объект `Graphics` посредством обращения к методу `CreateGraphics()`. Когда мы получаем объект `Graphics` от события `OnPaint()`, то этот объект создается не нами, и, соответственно, ответственность за вызов метода `Dispose()` также лежит не на нас, а в предшествующем примере эта ответственность возлагается именно на нас.

Вызов метода `Dispose()` выполняется автоматически в деструкторе различных классов, в которых реализуется интерфейс `IDisposable`. Возможно, вы подумали, что это освобождает вас от ответственности за вызов метода `Dispose()`, однако это не так. Причина такого положения вещей заключается в том, что деструктор может быть вызван только сборщиком мусора (GC, garbage collector), а заранее никогда не известно, в какой момент GC будет запущен. В частности, на операционной системе Windows 9X, использующей большие объемы памяти, GC может запускаться очень редко, и все ресурсы могут прекрасно использоваться до того, как GC будет запущен. Исчерпание памяти приводит к запуску GC, а вот исчерпание других ресурсов — нет. Система Windows 2000 оказывается гораздо менее чувствительной к исчерпанию ресурсов. В соответствии со спецификациями в Windows 2000 не существует каких-либо конкретных ограничений на эти типы ресурсов; автору, тем не менее, приходилось сталкиваться с ситуацией, когда эта операционная система начинала работать неправильно из-за большого количества открытых в ней приложений, при этом закрытие нескольких приложений приводило к быстрому восстановлению работоспособности системы. В любом случае, более правильным стилем программирования является удаление всех "жадных до ресурсов" объектов на регулярной основе.

В версии .NET Framework Beta предусмотрена новая конструкция, которая может использоваться для принудительного обращения к методу `Dispose()`. Конструкция `using` автоматически запускает метод `Dispose()`, как только происходит выход из области действия данного объекта. Следующий код представляет собой пример правильного использования ключевого слова `using` в этом контексте:

```
using (Graphics g = this.CreateGraphics())
{
   g.DrawLine(Pens.Black, new Point(0, 0),
      new Point(3, 5));
}
```

В соответствии с документацией этот код абсолютно эквивалентен следующему:

```
Graphics g = this.CreateGraphics();
try {
   g.DrawLine(Pens.Black, new Point(0, 0),
      new Point(3, 5));
}
finally {
   if (g != null)
      ((IDisposable)g).Dispose();
}
```

Не следует путать использование ключевого слова `using` в этом контексте с использованием его в качестве директивы, которая приводит к созданию нового имени пространства имен или позволяет использовать типы, описанные в некотором пространстве имен, без необходимости применять полностью квалифицированные имена. Это совершенно особый случай использования ключевого слова `using`.

В примерах, приводимых в настоящей главе, используются оба способа обращения к методу `Dispose()`. Иногда мы будем вызывать метод `Dispose()` непосредственно, а иногда прибегать к использованию конструкции `using`. Второй способ — это гораздо более наглядное решение, что видно из приведенных выше отрывков программ, хотя не существует никаких рекомендаций относительно того, какой из этих двух способов является более предпочтительным.

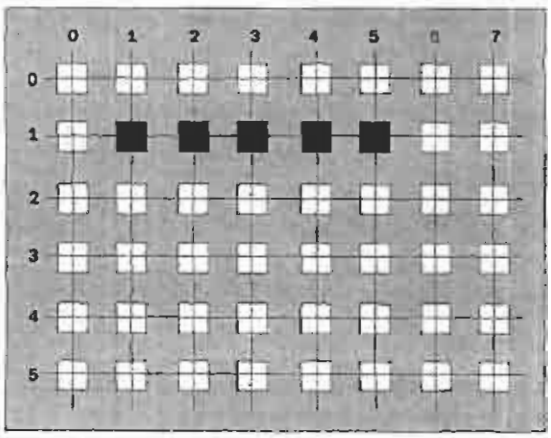
Перед тем как перейти к созданию первого примера, нам необходимо изучить два других аспекта рисования — систему координат и цвета.

Система координат

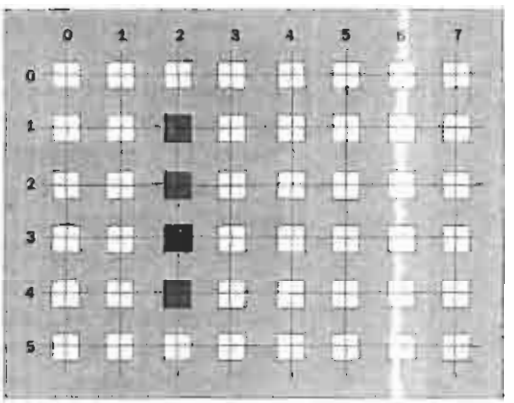
При разработке программы, которая должна осуществлять вывод сложной графики, представляется очень важным, чтобы код рисовал только то, что необходимо, и ничего больше. Даже один нарисованный в неправильном месте пиксель может оказать негативное влияние на восприятие целого изображения, поэтому необходимо понимать, какие именно пиксели будут рисоваться при выполнении тех или иных графических операций. Особую важность это имеет при создании собственных управляющих элементов, которые обычно состоят из большого количества прямоугольников, а также вертикальных и горизонтальных линий. Стоит при рисовании линии захватить один лишний пиксель или наоборот — нарисовать линию на один пиксель короче, как это тут же становится очень заметным. Такие ошибки несколько менее заметны в случае вывода кривых, диагоналей и некоторых других графических элементов.

В GDI+ используется система координат, построенная на основе воображаемых математических прямых, которые проходят через центры пикселей. Эти прямые пронумерованы начиная с 0: пересечение этих прямых в левом верхнем пикселе имеет координаты X = 0, Y = 0 во всех координатных пространствах.

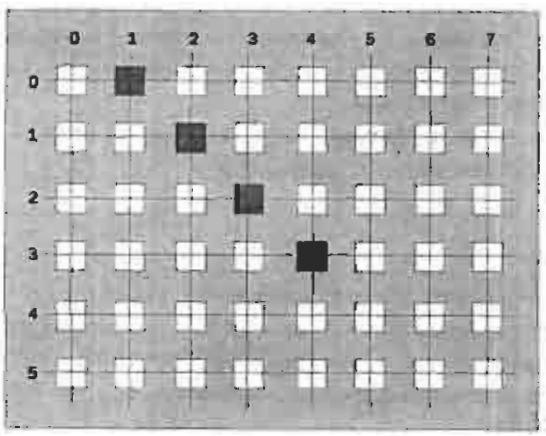
В упрощенной нотации можно говорить о точке (1, 2), что является сокращенным вариантом записи X = 1, Y = 2. Каждое окно, в котором будут выводиться какие-либо графические объекты, обладает своим координатным пространством. Если создать собственный управляющий элемент, который в дальнейшем будет использоваться в других окнах, то такой элемент также будет обладать своей системой координат. Другими словами, при выводе графических объектов в этом элементе его левый верхний угол будет иметь координаты 0, 0. И неважно, в каком месте содержащего его окна он будет располагаться.



При рисовании прямых GDI+ центрирует пиксели, находящиеся на заданной математической прямой. Рисование горизонтальной прямой с целыми координатами можно представить себе как проведение воображаемой математической линии таким образом, что половина пикселя оказывается расположенной выше нее, а вторая половина — ниже. Для рисования горизонтальной прямой, толщина которой составляет один пиксель и которая начинается в точке 1, 1 и заканчивается в точке 1, 5, будут использованы пиксели, закрашенные на рисунке слева черным.

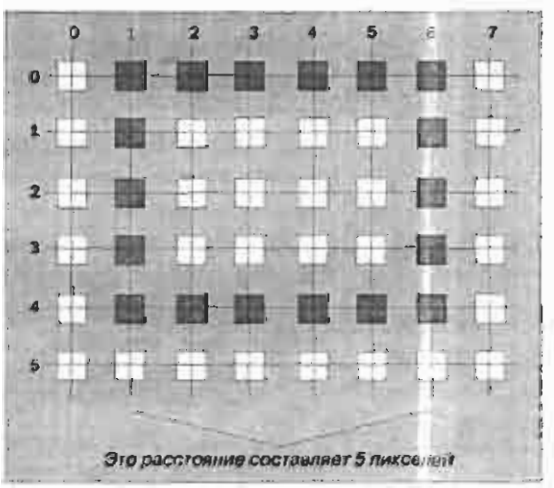


При рисовании вертикальной прямой толщиной в один пиксель и длиной в четыре пикселя из точки 2, 1 в точку 2, 4 будут закрашены пиксели, как на рисунке справа.



Если мы рисуем диагональную линию между точками 1, 0 и 4, 3, то закрашенными окажутся пиксели, как показано на рисунке слева.

Прямоугольник, верхний левый угол которого располагается в точке 1, 0 и который имеет размеры 5, 4, будет нарисован так, как это изображено на рисунке справа.



Заметим, что ширина прямоугольника была задана равной 5 пикселям, но на рисунке в горизонтальном направлении оказались закрашенными 6 пикселей. Однако если подсчитать количество математических прямых, проходящих через составляющие заданный прямоугольник пиксели, то можно убедиться, что

ширина прямоугольника составляет как раз пять пикселей, а проведенная линия оставляет одну половину пикселя снаружи заданной математической прямой, а вторую — внутри.

Если линия будет рисоваться в режиме сглаживания, то некоторые пиксели будут закрашиваться только наполовину, создавая иллюзию непрерывной линии и позволяя частично избавиться от представления диагональных линий в виде "лесенки".



Вот как выглядит линия, нарисованная без использования сглаживания (см. рис. слева).



А на рисунке справа показана та же линия, нарисованная с использованием сглаживания.

При выводе с большим разрешением эта линия будет выглядеть гораздо более плавной, без "лесенки".

Понимание взаимосвязи между координатами, передаваемыми графическим функциям, и эффектом, к появлению которого на поверхности рисования это приведет, позволяет точно представлять, какие именно пиксели будут задействованы при данном вызове графической функции.

Существуют три структуры, которые мы будем наиболее часто использовать для задания координат при рисовании: `Point` (точка), `Size` (размер) и `Rectangle` (прямоугольник).

Point

В GDI+ `Point` используется для представления отдельной точки. Это точка, расположенная на двумерной плоскости, которая определяет единственный пиксель. Многим функциям, использующимся в GDI+, таким как `DrawLine()`, `Point` передается в качестве аргумента. Мы объявляем и определяем точку следующим образом:

```
Point p = new Point(1, 1);
```

У точки `Point` имеются общие свойства, которые позволяют узнавать и задавать ее координаты *X* и *Y*.

Size

`Size` в GDI+ используется для представления размера, выраженного в пикселях. Структура `Size` определяет как ширину, так и высоту. Мы объявляем и определяем `Size` следующим образом:

```
Size s = new Size(5, 5);
```

Узнавать и задавать ширину и высоту, определяемую структурой `Size`, можно с помощью соответствующих общих свойств.

Rectangle

Эта структура используется в GDI+ во многих различных местах для задания координат прямоугольников. Структура `Point` описывает верхний левый угол прямоугольника, а структура `Size` — его размеры. У `Rectangle` есть два конструктора. Одному конструктору в качестве аргументов передаются координаты *X*, *Y* — ширина

и высота. Второй конструктор принимает структуры `Point` и `Size`. Ниже приводятся два примера объявления и создания структуры `Rectangle`:

```
Rectangle r1 = new Rectangle(1, 2, 5, 6);

Point p = new Point(1, 2);
Size s = new Size(5, 6);
Rectangle r2 = new Rectangle(p, s);
```

Для определения и задания всех составляющих местоположения и размеров прямоугольника используются общие свойства. Кроме того, существуют и другие свойства и методы, которые позволяют осуществлять разные полезные вещи, например, проверять является ли прямоугольник пустым, определять, пересекается данный прямоугольник с каким-либо другим прямоугольником, а также получать пересечение и объединение двух прямоугольников. Подробнее эти вопросы освещены в разделе, посвященном структуре `Rectangle` в .NET Framework Reference (руководство по .NET Framework).

Структуры GraphicsPath

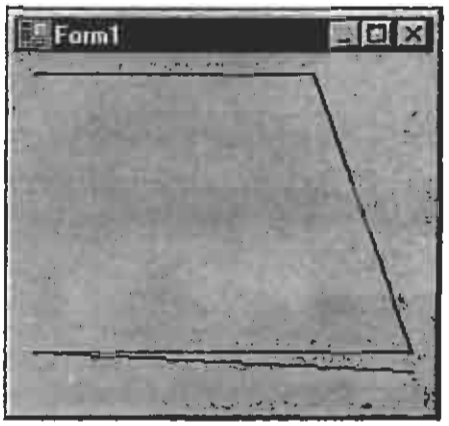
Есть два других, более важных типа данных, которые могут использоваться в качестве аргументов, передаваемых различным функциям графического вывода в GDI+. Класс `GraphicsPath` (графический путь) представляет собой последовательность соединенных между собой прямых и кривых. Для создания графического пути могут использоваться прямые, кривые Безье, дуги, секторы, многоугольники, прямоугольники и многое другое. После определения сложного пути можно вывести его на экран посредством одной операции — обратившись к методу `DrawPath()`. Для наполнения пути можно использовать метод `FillPath()`.

Для создания структуры `GraphicsPath` используется массив точек и массив `PathTypes` — массив байтов, в котором каждый элемент соответствует элементу из массива точек и содержит информацию о том, как следует строить графический путь через данную точку. Например, если точка является начальной точкой пути, то тип пути для этой точки будет иметь значение `PathPointType.Start`. Если эта точка является точкой соединения двух линий, то тип пути для этой точки будет иметь значение `PathPointType.Line`. Если точка используется для создания кривой Безье между двумя точками, то тип пути для этой точки будет иметь значение `PathPointType.Bezier`.

Ниже приводится пример программы, которая создает графический путь, состоящий из четырех прямых участков:

```
GraphicsPath path;
path = new GraphicsPath(new Point[]{
                        new Point(10, 10),
                        new Point(150, 10),
                        new Point(200, 150),
                        new Point(10, 150),
                        new Point(200, 160)
                                    }, new byte[] {
                                            (byte)PathPointType.Start,
                                            (byte)PathPointType.Line,
                                            (byte)PathPointType.Line,
                                            (byte)PathPointType.Line,
                                            (byte)PathPointType.Line,
                                              }
                        );
e.Graphics.DrawPath(Pens.Black, path);
```

Линия, которая будет нарисована с помощью данного пути, показана на рисунке справа.



Код, понадобившийся для создания такого пути, оказывается достаточно сложным. Конструктор `GraphicsPath` принимает два аргумента. Первый из них — это массив структур `Point`, создаваемый непосредственно здесь же. Для этого используется принятый в C# синтаксис, позволяющий объявлять и инициализировать массивы. Мы последовательно вызываем конструкторы для каждой точки:

```
new Point[]{
            new Point(10, 10),
            new Point(150, 10),
            new Point(200, 150),
            new Point(10, 150),
            new Point(200, 160)
           }
```

В качестве второго аргумента используется массив байтов, который создается во вторую очередь:

```
new byte[] {
            (byte)PathPointType.Start,
            (byte)PathPointType.Line,
            (byte)PathPointType.Line,
            (byte)PathPointType.Line,
            (byte)PathPointType.Line,
           }
```

В завершение, в приведенном выше примере мы вызываем метод `DrawPath()`:

```
e.Graphics.DrawPath(Pens.Black, path);
```

Области

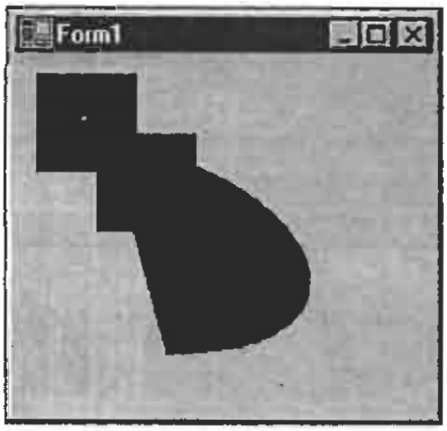
Класс `Region` представляет собой сложную графическую форму, которая состоит из прямоугольников и путей. После того как структура `Region` создана, она может быть выведена с помощью метода `FillRegion()`.

Следующая программа создает область, добавляет в нее сначала `Rectangle`, затем `GraphicsPath`, после чего заполняет эту область синим цветом:

```
Rectangle r1 = new Rectangle(10, 10, 50, 50);
Rectangle r2 = new Rectangle(40, 40, 50, 50);
Region r = new Region(r1);
r.Union(r2);

GraphicsPath path = new GraphicsPath(new Point[]{
     new Point(45, 45),
     new Point(145, 55),
     new Point(200, 150),
     new Point(75, 150),
     new Point(45, 45)
     }, new byte[] {
        (byte)PathPointType.Start,
        (byte)PathPointType.Bezier,
        (byte)PathPointType.Bezier,
```

```
                (byte)PathPointType.Bezier,
                (byte)PathPointType.Line,
                ));
            r.Union(path);
            e.Graphics.FillRegion(Brushes.Blue, r);
```



Этот код позволяет нарисовать форму, показанную на рисунке слева.

Код, необходимый для создания области, оказывается достаточно сложным, что связано с построением путей, которые потом пройдут через данную область. Создание области включает в себя построение прямоугольников и путей с последующим вызовом метода `Union()` (объединение). Если требуется изобразить пересечение пути и прямоугольника, то можно использовать вместо метода `Union()` метод `Intersection()` (пересечение).

Более подробная информация о графических путях и областях не является необходимой для ознакомления с GDI+. Ее можно почерпнуть в .NET Framework Reference (Руководство по .NET Framework).

Цвета

Большая часть графических операций в GDI+ предполагает выбор некоторого цвета. Например, при рисовании линии или прямоугольника необходимо указывать, каким цветом они должны рисоваться.

В GDI+ цвета инкапсулированы в структуру `Color`. Можно синтезировать цвет, передавая значения составляющих его красного, зеленого и синего цветов некоторой функции структуры `Color`, но необходимость в этом возникает крайне редко. В структуре `Color` содержится приблизительно 150 различных свойств, которые предоставляют широкий выбор заранее определенных цветов. Забудьте о красном, зеленом, синем, желтом и черном! Если требуется вывести какой-либо графический объект цветом `LightGoldenrodYellow` (светло-золотисто-желтый) или цветом `LavenderBlush` (бледно-лилово-розовый), то такие цвета заранее описаны специально для нас! Нам необходимо описать переменную типа `Color` и присвоить ей в качестве начального значения цвет, описанный в структуре `Color` таким образом:

```
Color redColor = Color.red;
Color anotherColor = Color.LightGoldenrodYellow;
```

Мы уже практически готовы к тому, чтобы нарисовать что-нибудь, однако прежде необходимо сделать пару замечаний.

Цвета могут представляться в двух различных видах. Один из них — это RGB. Второй заключается в разбиении цвета на три компонента — Saturation (насыщенность), Hue (оттенок) и Brightness (яркость). В структуре `Color` имеются специальные методы, которые позволяют осуществлять такое разбиение,— `GetHue()`, `GetSaturation()` и `GetBrightness()`.

Для того чтобы поэкспериментировать с цветами, можно воспользоваться приложением Paint. Выберите в меню пункт **Color | Edit Colors**. Нажмите кнопку **Define Custom Colors** (определение собственных цветов), и перед вами появится диалоговое окно, которое позволит выбирать цвет с помощью мыши и получать для него соответствующие значения RGB. Существует также возможность получить для выбранного цвета значения Hue, Saturation и Luminosity (Luminosity в данном

случае — это то же самое, что Brightness). Кроме того, есть возможность непосредственно вводить значения RGB и наблюдать получаемый в результате цвет.

Цвета в GDI+ обладают еще четвертым компонентом — компонентом Alpha, с помощью которого можно задавать затененность цвета, что позволяет создавать эффекты "из затемнения"/"в затемнение". Такие эффекты используются в меню Windows 2000. В данной главе использование компонента Alpha рассматриваться не будет.

Вывод линий с помощью класса Pen

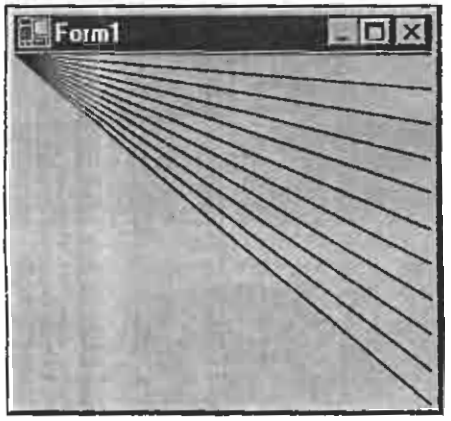
Первый пример будет посвящен рисованию линий. Мы будем рисовать линии с помощью класса `Pen`, определяющего цвет, толщину и образец линии, которая рисуется программой. Предназначение свойств Color (цвет) и Width (толщина) очевидны, а образец позволяет определять, требуется ли нарисовать сплошную линию или линию, состоящую из черточек и точек. Класс `Pen` находится в пространстве имен `System.Drawing`.

Практикум: Пример использования класса Pen

Запустите Visual Studio.NET и создайте новое C# Windows Application. При создании нового проекта Visual Studio.NET создает также и новую пустую форму для этого проекта, которая получает имя `Form1`. В модуле `Form1.cs` содержится код, предназначенный для этой формы. Щелкните на форме правой кнопкой мыши и выберите пункт View Code из открывшегося меню. В качестве альтернативы можно щелкнуть правой кнопкой мыши на `Form1.cs` в Solution Explorer, чтобы выбрать пункт View Code из открывшегося меню. Добавьте в `Form1.cs` следующий код:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;

   using (Pen blackPen = new Pen(Color.Black, 1))
   {
      for (int y = 0; y < ClientRectangle.Height;
                        y += ClientRectangle.Height / 10)
      {
         g.DrawLine(blackPen, new Point(0, 0),
                    new Point(ClientRectangle.Width, y));
      }
   }
}
```



Теперь нажмите *F5* и запустите программу. В результате работы этой программы будет создано окно, представленное на рисунке слева.

Как это работает

В приведенной выше программе в самом начале метода создается экземпляр класса `Graphics`, для инициализации которого используется класс `PaintEventArgs`:

```
Graphics g = e.Graphics;
```

Поскольку мы получаем ссылку на объект `Graphics` и нам не приходится создавать его самим, то нет необходимости явно вызывать для него метод `Dispose()`. Поскольку в данном

примере мы используем потенциально потребляющий много ресурсов объект Pen, то весь остальной код мы помещаем внутрь конструкции using, что гарантирует уничтожение объекта при первой же возможности.

При создании объекта Pen мы передаем его конструктору в качестве параметров цвет и ширину линии. В данном примере в качестве цвета выбран черный, а ширина устанавливается равной единице. Вот строка кода, предназначенная для создания этого объекта:

```
using (Pen blackPen = new Pen(Color.Black, 1))
```

Каждое окно, в котором что-либо рисуется, обладает пользовательской областью, обрамляемой границей и в точности соответствующей области, в которой может располагаться рисунок. Получить пользовательскую область можно с помощью ClientRectangle — общего свойства формы, доступного в режиме "только чтение" (которое наследуется от класса Control). В нем содержатся размеры (ширина и высота) пользовательской области окна, в которую осуществляется графический вывод. Следующий код обращается к свойству ClientRectangle:

```
for (int y = 0; y < ClientRectangle.Height;
                y += ClientRectangle.Height / 10)
```

При рисовании каждой линии только что созданному объекту Pen передаются начальная точка и конечная точка этой линии:

```
g.DrawLine(blackPen, new Point(0, 0),
           new Point(ClientRectangle.Width, y));
```

Всегда вызывайте метод Dispose() для объектов Pen.

Как и для объектов Graphics, необходимо либо вызывать метод Dispose() для объектов Pen после завершения его использования, либо использовать для работы с ними конструкцию using, в противном случае приложение может исчерпать ресурсы системы.

В данном примере мы используем объект Pen. Однако существует более простая возможность получить такой объект. Класс Pens содержит свойства, необходимые для получения приблизительно 150 объектов Pen — по одному на каждый из предварительно описанных цветов. Следующая версия нашего примера работает аналогично предшествующей, только вместо создания объекта Pen мы извлекаем его из класса Pens:

```
protected override void OnPaint(PaintEventArgs e)
{
   for (int y = 0; y < ClientRectangle.Height;
                   y += ClientRectangle.Height / 10)
   {
      e.Graphics.DrawLine(Pens.Black, new Point(0, 0),
                          new Point(ClientRectangle.Width, y));
   }
}
```

В данном случае объект Pen создается не нами, поэтому нет необходимости вызывать метод Dispose().

Рисование фигур с помощью класса Brush

Следующий пример посвящен использованию класса `Brush` (кисть) для рисования различных фигур, таких как прямоугольники, эллипсы, секторы и многоугольники. Класс `Brush` — это абстрактный класс. Для создания экземпляра класса `Brush` мы используем классы, производные от класса `Brush`, такие как `SolidBrush`, `TextureBrush` и `LinearGradientBrush`. Класс `Brush` находится в пространстве имен `System.Drawing`. Классы `TextureBrush` и `LinearGradientBrush` находятся в пространстве имен `System.Drawing.Drawing2D`. Вот что позволяет делать каждый из этих классов:

- `SolidBrush` заполняет фигуру сплошным цветом.
- `TextureBrush` позволяет заполнять фигуру рисунком, хранящемся в двоичном представлении. При создании такой кисти требуется также задавать обрамляющий прямоугольник и режим обрамления. Обрамляющий прямоугольник определяет, какую порцию рисунка мы должны использовать при рисовании,— использовать весь рисуиок целиком совершенно необязательно. Для режима обрамления существует несколько возможностей, включая `Tile` (черепица) — `TileFlipX`, `TileFlipY` и `TileFlipXY`, позволяющих последовательно разбивать изображение на отдельные квадраты. С помощью `TextureBrush` можно создавать очень интересные и весьма впечатляющие эффекты.
- `LinearGradientBrush` содержит кисть, которая позволяет рисовать плавный переход от одного цвета к другому, причем первый цвет переходит во второй под определенным углом. Углы при этом задаются в градусах. Угол, равный 0°, означает, что переход от одного цвета к другому осуществляется слева направо. Угол, равный 90°, означает, что переход от одного цвета к другому осуществляется сверху вниз.

Существует еще одна разновидность кисти — `PathGradientBrush`, позволяющая создавать сложный эффект затенения, при котором используется изменение цвета от середины рисуемого пути к его краям. Более детально ознакомиться с этим классом вы можете в разделе `PathGradientBrush` .NET Framework Reference (руководство по .NET Framework).

Всегда вызывайте метод `Dispose()` для объектов `Brush`.

Как и для объектов `Graphics` и `Pens`, необходимо либо вызывать метод `Dispose()` для создаваемых нами объектов `Brush`, либо использовать для работы с ними конструкцию `using`, в противном случае приложение может исчерпать ресурсы системы.

Практикум: пример использования класса Brush

Запустите Visual Studio.NET и создайте новое C# Windows Application. Щелкните на форме правой кнопкой мыши и выберите пункт View Code из открывшегося меню. Найдите конструктор класса `Form1` и добавьте в него обращение к методу `SetStyle()` сразу за обращением к методу `InitializeComponent()`. `SetStyle()` —

это функция класса Form. Видоизмененный конструктор будет выглядеть следующим образом:

```
public Form1()
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();
   SetStyle(ControlStyles.Opaque, true);

   //
   // Следует сделать: Любой код, относящийся к конструктору,
   // должен быть помещен после вызова метода InitializeComponent
   //
}
```

Этот код изменяет поведение класса Form таким образом, что он не будет автоматически выполнять закрашивание фона окна. Если мы включаем в код эту строку, но не закрашиваем фон окна самостоятельно, то какое бы изображение ни оказалось расположенным под данным окном на момент его создания, оно будет оставаться видимым, что вряд ли может нас устроить.

В самое начало файла мы включим строку с оператором using, поскольку в данном примере мы будем использовать класс LinearGradientBrush, который находится в пространстве имен Drawing2D. После добавления директивы using к пространству имен Drawing2D, код примет следующий вид:

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

Теперь необходимо добавить к нашему классу метод OnPaint():

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.FillRectangle(Brushes.White, ClientRectangle);
   g.FillRectangle(Brushes.Red, new Rectangle(10, 10, 50, 50));

   Brush linearGradientBrush = new LinearGradientBrush(
         new Rectangle(10, 60, 50, 50), Color.Blue, Color.White, 45);
   g.FillRectangle(linearGradientBrush, new Rectangle(10, 60, 50, 50));

   // Вызываем метод Dispose() вручную
   linearGradientBrush.Dispose();

   g.FillEllipse(Brushes.Aquamarine, new Rectangle(60, 20, 50, 30));
   g.FillPie(Brushes.Chartreuse, new Rectangle(60, 60, 50, 50), 90, 210);
   g.FillPolygon(Brushes.BlueViolet, new Point[] {
                                          new Point(110, 10),
                                          new Point(150, 10),
                                          new Point(160, 40),
                                          new Point(120, 20),
                                          new Point(120, 60),
                                          });
}
```



Теперь нажмите клавишу *F5* и запустите программу. В результате ее выполнения будет создано окно, представленное на рисунке слева.

Как это работает

Существует класс `Brushes`, в котором содержится приблизительно 150 разновидностей кистей — по одной на каждый заранее определенный цвет. Названный класс будет использоваться нами для создания большинства кистей, с которыми мы будем работать в данном примере, за исключением кисти `LinearGradientBrush`, которую мы создадим самостоятельно.

Первое обращение к методу `FillRectangle()` позволяет нарисовать фон клиентской области нашего окна:

```
g.FillRectangle(Brushes.White, ClientRectangle);
```

Создание `LinearGradientBrush` предполагает передачу прямоугольника с заданными размерами, двух цветов, между которыми будет осуществляться переход, и угол этого перехода, который в данном случае равен 45°:

```
Brush linearGradientBrush = new LinearGradientBrush(
            new Rectangle(10, 60, 50, 50), Color.Blue, Color.White, 45);
g.FillRectangle(linearGradientBrush, new Rectangle(10, 60, 50, 50));
linearGradientBrush.Dispose();
```

Когда мы задаем прямоугольник для кисти, то используем прямоугольник с шириной, равной 50, и высотой, равной 50, что в точности совпадает с прямоугольником, использованным при описании кисти. В результате этого площадь кисти соответствует закрашиваемой области. Попробуйте изменить размеры прямоугольника, описываемого при создании кисти, таким образом, чтобы его ширина и высота были равны 10, и посмотрите, что получится. Кроме того, попытайтесь использовать для угла изменения цвета (который в вышеприведенной программе описан как равный 45°) различные другие значения, и понаблюдайте, каким образом изменится эффект.

Графический вывод текста с помощью класс Font

Следующий пример будет посвящен графическому выводу текста с помощью класса `Font`, который включает в себя три основные характеристики шрифта, а именно: семейство, размер и стиль. Класс `Font` находится в пространстве имен `System.Drawing`.

Согласно документации .NET, семейство шрифтов "определяет группу типов представления литер, имеющих одинаковый базовый дизайн". Это довольно причудливый способ объяснить, что Courier, Arial и Times New Roman представляют собой семейства шрифтов.

Свойство `Size` определяет размер данного типа шрифта. Однако в .NET Framework, строго говоря, это свойство не является обязательно размером шрифта, выраженным в пунктах. Оно может представлять собой такой размер, однако существует возможность изменить свойство с названием `GraphicsUnit` (единица измерения графических объектов) посредством свойства `Unit`, которое определяет единицу измерения шрифтов. Один пункт составляет $^{1}/_{72}$ дюйма, поэтому шрифт в 10 пунктов будет иметь высоту $^{10}/_{72}$ дюйма. С помощью перечислимого типа

`GraphicsUnit` размер шрифта может задаваться с помощью следующих единиц измерения:

- пункты
- особый шрифт ($^1/_{75}$ дюйма)
- документ ($^1/_{300}$ дюйма)
- дюйм
- миллиметр
- пиксель

Это означает, что мы обладаем беспрецедентной гибкостью при задании требуемого размера шрифта. Одной из возможных областей применения этих размеров может быть подпрограмма, предназначенная для графического вывода текста, которая должна надлежащим образом работать с дисплеями, обладающими высоким/низким разрешением и с принтерами.

При выводе в графическом режиме текста каким-либо определенным шрифтом на некоторой конкретной поверхности рисования зачастую бывает необходимо узнать длину данной строки текста, выраженную в пикселях. Совершенно очевидно, почему в результате использования различных шрифтов длина строки, выраженная в пикселях, будет различной — меньший по размеру шрифт будет занимать меньшее количество пикселей. Однако не менее важно обладать информацией о поверхности рисования, поскольку разрешение различных поверхностей рисования, выраженное в пикселях, также различно. Обычно экран имеет разрешение 72 пикселя на дюйм. У некоторых принтеров может быть разрешение 300 или 600 пикселей на дюйм, а иногда даже еще больше. Для определения длины строки для данного шрифта мы будем использовать метод `MeasureString()` объекта `Graphics`. В качестве образца здесь использован некоторый код, который позволяет продемонстрировать, каким образом можно получить длину и высоту строки. После этого код выводит прямоугольник черного цвета, внутри которого синим цветом выводится текст:

```
String str = "This is a string";
SizeF size = g.MeasureString(str, Font);
g.DrawRectangle(Pens.Black, 0, 0, size.Width, size.Height);
g.DrawString(str, Font, Brushes.Blue, new RectangleF(0, 0,
                size.Width, size.Height));
```

Свойство шрифта `Style` (стиль) позволяет определять, выделяется ли текст курсивом, жирным шрифтом, зачеркивается ли он или подчеркивается.

Всегда вызывайте метод `Dispose()` для объектов `Font`.

Вызов метода `Dispose()` для создаваемых нами объектов `Brush` или использование для работы с ними конструкции `using` представляется очень важным, поскольку в противном случае приложение может исчерпать ресурсы системы.

При графическом выводе текста для задания граничных координат выводимого текста используется `Rectangle`. Обычно высота этого прямоугольника должна равняться высоте шрифта или быть кратной ей. Отступление от этого правила допускается только в особых случаях для достижения специальных эффектов с использованием отдельных частей текста.

В классе StringFormat содержится информация о внешнем представлении текста, включая выравнивание и интервалы между строками. В следующем примере демонстрируется выравнивание текста по правому краю и по центру с помощью класса StringFormat.

Практикум: пример использования класса Font

Запустите Visual Studio.NET и создайте проект C# Windows Application. Щелкните правой кнопкой мыши на форме и выберите пункт View Code из раскрывшегося меню. Найдите конструктор класса Form1 и вставьте обращение к методу SetStyle() сразу после обращения к методу InitializeComponent(). Нам также потребуется изменить границы окна, что бы в нем было достаточно места для вывода текста, который нам необходим. Измененный конструктор будет выглядеть следующим образом:

```
public Form1()
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();
   SetStyle(ControlStyles.Opaque, true);
   Bounds = new Rectangle(0, 0, 500, 300);

   //
   // Следует сделать: Любой код, относящийся к конструктору,
   // должен быть помещен после вызова метода InitializeComponent
   //
}
```

Теперь добавьте в класс метод OnPaint():

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;
   int y = 0;

   g.FillRectangle(Brushes.White, ClientRectangle);

   // Вывод текста, выровненного по левому краю
   Rectangle rect = new Rectangle(0, y, 400, Font.Height);
   g.DrawRectangle(Pens.Blue, rect);
   g.DrawString("This text is left justified.", Font,
                Brushes.Black, rect);
   y += Font.Height + 20;

   // Вывод текста, выровненного по правому краю
   Font aFont = new Font("Arial", 16, FontStyle.Bold | FontStyle.Italic);
   rect = new Rectangle(0, y, 400, aFont.Height);
   g.DrawRectangle(Pens.Blue, rect);

   StringFormat sf = new StringFormat();
   sf.Alignment = StringAlignment.Far;
   g.DrawString("This text is right justified.", aFont, Brushes.Blue,
                rect, sf);
   y += aFont.Height + 20;

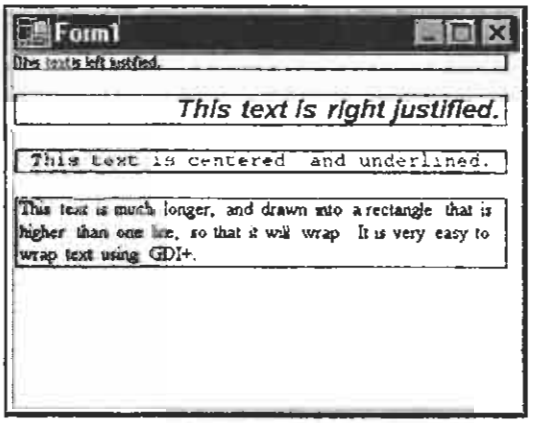
   // Вызов удаления текста вручную
   aFont.Dispose();
```

```
        // Вывод текста, выровненного по центру
        Font cFont = new Font("Courier New", 12, FontStyle.Underline);
        rect = new Rectangle(0, y, 400, cFont.Height);
        g.DrawRectangle(Pens.Blue, rect);
        sf = new StringFormat();
        sf.Alignment = StringAlignment.Center;
        g.DrawString("This text is centered and underlined.", cFont,
                     Brushes.Red, rect, sf);
        y += cFont.Height + 20;

        // Вызов удаления текста вручную
        cFont.Dispose();

        // Вывод многострочного текста
        Font trFont = new Font("Times New Roman", 12);
        rect = new Rectangle(0, y, 400, trFont.Height * 3);
        g.DrawRectangle(Pens.Blue, rect);
        String longString = "This text is much longer, and drawn ";
        longString += "into a rectangle that is higher than ";
        longString += "one line, so that it will wrap. It is ";
        longString += "very easy to wrap text using GDI+.";
        g.DrawString(longString, trFont, Brushes.Black, rect);

        // Вызов удаления текста вручную
        trFont.Dispose();
    }
```



Теперь нажмите *F5*, чтобы откомпилировать и запустить программу. Запуск программы приведет к созданию окна, показанного на рисунке слева.

Как это работает

В этом примере содержится несколько наиболее распространенных операций вывода текста в графическом режиме.

Для удобства мы присваиваем ссылку на объект `Graphics` локальной переменной и закрашиваем фон окна белым цветом.

При рисовании окна мы вычисляем размеры прямоугольника, необходимые для размещения текста. Высота шрифта определяется с помощью свойства `Height`. Для наглядности этот прямоугольник рисуется голубым цветом, что позволяет четко выделить прямоугольник, ограничивающий текст. При рисовании мы передаем сам текст, шрифт, тип кисти и ограничивающий прямоугольник:

```
// Вывод текста, выровненного по левому краю
Rectangle rect = new Rectangle(0, y, 400, Font.Height);
g.DrawRectangle(Pens.Blue, rect);
g.DrawString("This text is left justified.", Font,
             Brushes.Black, rect);
```

Нами определяется только тот прямоугольник, в котором будет размещаться текст. Основная линия шрифта — это воображаемая линия, на которой "сидят" почти все символы данного шрифта. GDI+ и сам шрифт определяют, где в действительности будет проходить эта линия. Мы не можем управлять этим процессом.

При рисовании текста мы передаем кисть функции `DrawString()`. В данном примере передаются только кисти, которые рисуют исключительно сплошным цветом. С тем же успехом мы могли бы передавать и другие разновидности кистей,

в частности, выполняющие рисование с переходом от одного цвета к другому. После того как в следующем разделе мы познакомимся с понятием образов, мы продемонстрируем вывод текста в графическом режиме с использованием TextureBrush.

Для вывода первой порции текста в этом примере берется шрифт, используемый формой по умолчанию. На этот шрифт ссылается свойство Font, которое наследуется от Control:

```
g.DrawString("This text is left justified.", Font,
             Brushes.Black, rect);
```

Для вывода следующей порции текста создадим новый экземпляр объекта Font:

```
Font aFont = new Font("Arial", 16, FontStyle.Bold | FontStyle.Italic);
```

В этом примере демонстрируется не только то, каким образом можно создать новый экземпляр шрифта, но и то, как ему можно придать соответствующий стиль. В данном случае мы используем выделение жирным шрифтом и курсивом.

Пример также демонстрирует создание объекта StringFormat, в результате чего появляется возможность рисовать текст, выровненный по правому краю или по центру. В GDI+ для выравнивания текста по правому краю используется термин Far (дальнее), для выравнивания по левому краю — термин Near (ближнее):

```
StringFormat sf = new StringFormat();
sf.Alignment = StringAlignment.Far;
```

Затем выводится некоторый многострочный текст. Нет ничего проще, чем выполнять такую операцию с использованием GDI+. Все, что для этого требуется,— задать прямоугольник, ширина которого меньше длины строки выводимого теста (в пикселях), а высота достаточна для того, чтобы в нем уместилось несколько строк. В данном случае высота прямоугольника превосходит высоту шрифта в три раза.

Рисование с использованием образов

Образы широко применяются в GDI+. Можно рисовать образы в окнах, однако также можно создать кисть (TextureBrush) с неким образом и рисовать с ее помощью различные фигуры, которые затем будут заполнены этим образом. Используя кисть TextureBrush, мы можем создавать объект Pen, который позволит использовать данный образ при рисовании текста. Класс Image находится в пространстве имен System.Drawing.

Другим очень важным способом применения образов является специальная техника графического программирования, называемая **двойной буферизацией**. Иногда рисунок, который необходимо создать, оказывается очень сложным и запутанным, и даже современным быстродействующим компьютерам может потребоваться для его создания значительное время. Вряд ли кому-нибудь будет интересно наблюдать эффект замедленного формирования на экране рисунка. Примерами такого рода могут служить графические приложения и сложные приложения CAD/CAM. При использовании этого способа процедура рисования осуществляется не в окне, а в некотором образе, и, только после того как рисование завершено, этот образ переносится в окно. Такая техника известна под названием двойной буферизации. Существуют и некоторые другие приемы рисования, например, способ послойного рисования, при котором сначала рисуется фон, затем поверх фона рисуются объекты, а затем поверх этих объектов выводится текст. Если рисование таким способом осуществляется непосредственно на экране, то пользователь наблюдает эффект изменяющейся картинки. Двойная буферизация позволяет избавиться от такого эффекта.

Класс `Image` является абстрактным. У него существует два класса-наследника: `Bitmap` и `Metafile`.

Класс `Bitmap` используется при создании многоцелевых образов, обладающих определенными высотой и шириной. В примерах, использованных в настоящем разделе, работа будет вестись с помощью класса `Bitmap`. Мы загрузим образ типа `Bitmap` из файла и нарисуем его. Мы также создадим на его базе кисть, на основе которой, в свою очередь, будет создан объект класса `Pen` для рисования линий, кроме того, мы воспользуемся созданной кистью для рисования некоторого текста.

Класс `Metafile` будет рассматриваться ниже в настоящей главе, где обсуждаются дополнительный возможности GDI+.

Всегда вызывайте метод `Dispose()` для объектов `Image`.

Очень важно либо вызывать метод `Dispose()` для создаваемых объектов `Image`, либо использовать для работы с ними конструкцию `using`, в противном случае приложение может исчерпать ресурсы системы.

Существует несколько источников получения растрового изображения: его можно загрузить из файла; оно может поступить из открытого потока; оно может быть создано из уже существующего образа; оно может быть создано как пустой образ, в котором потом будет осуществляться рисование. Если образ считывается из файла, то такой файл должен быть представлен в одном из следующих форматов: JPEG, GIF или BMP.

Практикум: пример использования класса Image

Приведенный здесь пример — очень простой; в нем производится считывание образа из файла и его рисование в окне. Запустите Visual Studio.NET и создайте новый проект C# Windows Application. Щелкните на форме правой кнопкой мыши и выберите в появившемся меню пункт View Code. Необходимо объявить частную переменную в классе `Form1`, в которой будет храниться используемый образ после того, как мы считаем его из файла. Добавьте описание переменной `theImage` сразу после описания переменной `components`, как показано ниже:

```
public class Form1 : System.Windows.Forms.Form
{
   /// <summary>
   /// Переменная, которая необходима для разработки.
   /// </summary>
   private System.ComponentModel.Container components;
   private Image theImage;
```

Внесите изменения в конструктор, который должен выглядеть таким образом:

```
public Form1()
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();
   SetStyle(ControlStyles.Opaque, true);
   theImage = new Bitmap("swipes.jpg");
   //
   // Следует сделать: Любой код, относящийся к конструктору,
   // должен быть добавлен после вызова метода InitializeComponent
   //
}
```

Теперь добавьте в наш класс метод `OnPaintEvent()`:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;

   g.DrawImage(theImage, ClientRectangle);
}
```

В заключение, нам необходимо уничтожить тот образ `Image`, который хранится в переменной класса. Модифицируйте метод `Dispose()` класса `Form1` следующим образом:

```
protected override void Dispose( bool disposing )
{
   if( disposing )
   {
      theImage.Dispose();
      if (components != null)
      {
         components.Dispose();
      }
   }
   base.Dispose( disposing );
}
```



Перед тем как создавать данный класс, требуется создать образ и поместить его в ту же директорию, в которой в дальнейшем будет строиться выполняемый модуль приложения. По умолчанию для этого используется поддиректория директории, в которой располагается проект Visual Studio. Например, если данный пример был нами создан в директории с именем `ImageExample`, то выполняемый модуль приложения будет строиться в директории `ImageExample\bin\Debug`. Именно в этой директории должен располагаться соответствующий JPG-файл.

Чтобы откомпилировать и запустить программу, нажмите клавишу *F5*. Запустив программу, вы увидите окно, представленное на рисунке слева.

Как это работает

В конструкторе создается экземпляр объекта `Bitmap`, который присваивается объявленной ранее переменной `Image`.

Далее в событии `OnPaint()` рисуется образ. При рисовании функции `DrawImage()` в качестве одного из аргументов передается соответствующий прямоугольник. Если размеры образа оказываются отличными от прямоугольника, то GDI+ автоматически изменяет размеры образа в соответствии с размерами прямоугольника. Одним из способов добиться того, чтобы GDI+ не изменяло размеров образа, является передача методу `DrawImage()` размеров рисунка, которые хранятся в свойствах `Width` и `Height`.

Рисование с помощью класса TextureBrush

Модифицируем пример таким образом, чтобы с помощью получаемого образа создавался объект `TextureBrush`, а затем с помощью этой кисти рисовался эллипс. Возьмем за основу предыдущий пример и внесем в него некоторые изменения:

Практикум: пример использования класса TextureBrush №1

Добавьте в класс Form1 еще одно объявление переменной типа Image:

```
public class Form1 : System.Windows.Forms.Form
{
   /// <summary>
   /// Переменная, которая необходима для разработки.
   /// </summary>
   private System.ComponentModel.Container components;
   private Image theImage;
   private Image smallImage;
```

В конструкторе из образа theImage мы создадим образ smallImage. При его создании мы зададим прямоугольник, размеры которого составляют одну десятую часть от размеров theImage. Это позволит получить меньший по размерам вариант исходного образа:

```
public Form1()
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();
   SetStyle(ControlStyles.Opaque, true);
   theImage = new Bitmap("swipes.jpg");
   smallImage = new Bitmap(theImage,
                  new Size(theImage.Width / 10, theImage.Height / 10));

   //
   // Следует сделать: Любой код, относящийся к конструктору,
   // должен быть добавлен после вызова метода InitializeComponent
   //
}
```

Замените описание события OnPaint() на следующее:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;

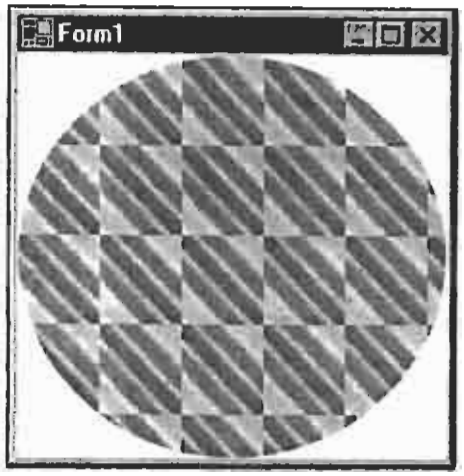
   g.FillRectangle(Brushes.White, ClientRectangle);

   Brush tBrush = new TextureBrush(smallImage, new Rectangle(0, 0,
                  smallImage.Width, smallImage.Height));
   g.FillEllipse(tBrush, ClientRectangle);
   tBrush.Dispose();
}
```

В заключение необходимо уничтожить оба образа, хранящихся в переменных класса. Модифицируйте метод Dispose() класса Form1 следующим образом:

```
protected override void Dispose( bool disposing )
{
   if( disposing )
   {
      theImage.Dispose();
      smallImage.Dispose();

      if (components != null)
      {
         components.Dispose();
      }
   }
   base.Dispose( disposing );
}
```



Запустив приложение, мы получим окно, показанное на рисунке слева.

Как это работает

При создании класса `TextureBrush` мы передаем конструктору некоторый прямоугольник, с помощью которого определяется, какая именно часть образа будет использоваться для создания кисти. В данном случае, мы задаем использование всего образа целиком. Теперь какой бы объект ни рисовался с помощью `TextureBrush`, он будет рисоваться с использованием двоичного представления, а не сплошного цвета.

Большая часть использованного в данном примере кода уже объяснялась по ходу главы. Единственное отличие заключается в том, что при вызове метода `FillEllipse()` класса `Graphics` мы передаем ему вновь созданный `TextureBrush`, и `ClientRectangle` рисует в окне эллипс:

```
g.FillEllipse(tBrush, ClientRectangle);
```

Рисование образов с помощью класса Pen

Практикум: пример использования класса TextureBrush №2

С помощью кисти `TextureBrush` можно создать соответствующую ручку. Возьмите за основу пример использования класса `TextureBrush` №1 и измените событие `OnPaint()` так:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;

   g.FillRectangle(Brushes.White, ClientRectangle);

   Brush tBrush = new TextureBrush(smallImage,
            new Rectangle(0, 0, smallImage.Width,
                          smallImage.Height));
   Pen tPen = new Pen(tBrush, 15);
   g.DrawRectangle(tPen, 20, 20, 240, 240);
   tPen.Dispose();
   tBrush.Dispose();
}
```



Запустив его, мы сможем увидеть окно, представленное на рисунке справа. Это пример того, каким образом можно рисовать с помощью ручки, которая была создана с использованием кисти, которая, в свою очередь, была создана с использованием образа.

Графический вывод текста с помощью класса Image

Практикум: пример использования класса TextureBrush №3

С помощью класса `TextureBrush` можно также выводить текст в графическом режиме. Внесите следующие изменения в событие `OnPaint()`:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;
```

```
        g.FillRectangle(Brushes.White, ClientRectangle);

        Brush tBrush = new TextureBrush(smallImage, new Rectangle(0, 0,
                                        smallImage.Width, smallImage.Height));
        Font trFont = new Font("Times New Roman", 32,
                               FontStyle.Bold | FontStyle.Italic);
        g.DrawString("Hello Image", trFont, tBrush, ClientRectangle);
        tBrush.Dispose();
        trFont.Dispose();
    }
```



Запустив измененный код, мы увидим окно, представленное на рисунке слева.

Вызов метода `DrawString()` аналогичен предыдущим вызовам этого метода. В качестве аргументов ему передается собственно текст, шрифт, созданная нами кисть и ограничивающий прямоугольник:

```
g.DrawString("Hello Image", trFont, tBrush,
ClientRectangle);
```

Двойная буферизация

Мы уже коротко касались проблем, которые возникают в тех случаях, когда процедура рисования требует слишком много времени и пользователю приходится очень долго ожидать, пока рисунок будет выведен на экран полностью. Как мы уже объясняли, решение этой проблемы заключается в создании образа, который затем — после завершения всех графических операций над ним — целиком выводится в окно.

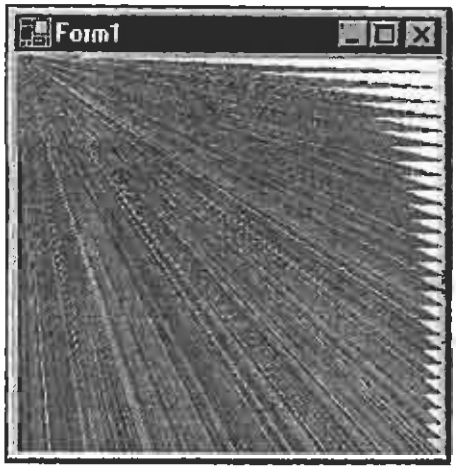
Практикум: пример без использования двойной буферизации

Использованная в данном примере программа `OnPaint()` рисует большое количество линий цветом, выбираемым случайным образом. Когда вы запускаете эту программу, то можете увидеть саму процедуру рисования (если, конечно, у вас не слишком быстродействующий компьютер):

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Random r = new Random();

    g.FillRectangle(Brushes.White, ClientRectangle);

    for (int x = 0; x < ClientRectangle.Width; x++)
    {
        for (int y = 0; y < ClientRectangle.Height; y += 10)
        {
            Color c = Color.FromArgb(r.Next(255), r.Next(255),
                                     r.Next(255));
            using (Pen p = new Pen(c, 1))
            {
                g.DrawLine(p, new Point(0, 0), new Point(x, y));
            }
        }
    }
}
```



Запустив программу и дождавшись окончания процедуры рисования, вы увидите в окне подобное представленному на рисунке слева.

Практикум: пример с использованием двойной буферизации

Если воспользоваться следующей версией процедуры `OnPaint()`, то весь рисунок будет выведен мгновенно после одно- или двухсекундной задержки:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics displayGraphics = e.Graphics;
   Random r = new Random();
   Image i = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
   Graphics g = Graphics.FromImage(i);

   g.FillRectangle(Brushes.White, ClientRectangle);

   for (int x = 0; x < ClientRectangle.Width; x++)
   {
      for (int y = 0; y < ClientRectangle.Height; y += 10)
      {
         Color c = Color.FromArgb (r.Next(255), r.Next(255),
                                   r.Next(255));
         Pen p = new Pen(c, 1);
         g.DrawLine(p, new Point(0, 0), new Point(x, y));
         p.Dispose();
      }
   }
   displayGraphics.DrawImage(i, ClientRectangle);
   i.Dispose();
}
```

Как это работает

В следующей строке мы создаем новый образ, обладающий той же высотой и шириной, что и `ClientRectangle`:

```
Image i = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
```

Далее, мы получаем объект `Graphics` из образа с помощью следующей строки:

```
Graphics g = Graphics.FromImage(i);
```

Все графические операции, которые выполняются,— те же самые, что и в предыдущем примере, единственное отличие заключается в том, что теперь их результат заносится в образ, а не выводится непосредственно в окне.

Наконец, перед завершением функции мы выводим образ в окне:

```
displayGraphics.DrawImage(i, ClientRectangle);
```

Поскольку все линии сначала заносятся в образ, который является невидимым, то прежде, чем мы сможем что-либо увидеть на экране, должно пройти некоторое время.

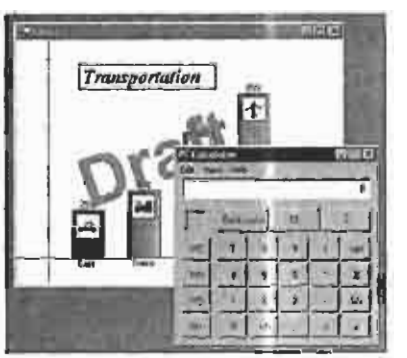
Дополнительные возможности GDI+

Мы только слегка затронули многие возможности, предоставляемые GDI+. На самом деле, его возможности намного шире — настолько, что их невозможно охватить в одной главе. Однако, прежде чем завершить эту главу, мы познакомимся с некоторыми из дополнительных возможностей.

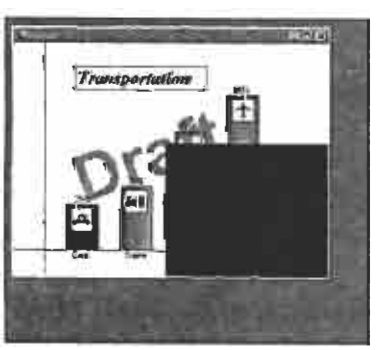
Вырезание отдельных частей рисунков

Существуют три области, для которых возможность работать с частями рисунков оказывается важной.

Во-первых, при вызове события `OnPaint()` ему кроме объекта `Graphics` передается вырезанный прямоугольник. При использовании простых процедур рисования можно не обращать особого внимания на этот вырезанный прямоугольник, однако если используется сложная процедура, требующая большого количества времени, то время рисования может быть уменьшено за счет проверки вырезанного прямоугольника перед выполнением соответствующих операций. Нам известен ограничивающий прямоугольник любого графического изображения или фигуры, которые мы рисуем. Если этот прямоугольник не пересекается с вырезанным, то можно не выполнять данную графическую операцию.



На рисунке справа представлен экран, где окно с диаграммой частично закрывается другим окном с калькулятором.



После того как калькулятор будет закрыт, а операционная система Windows восстановит границу оставшегося окна, окно с диаграммой примет вид, показанный на рисунке слева.

В этот момент будет вызвано событие `OnPaint()` для окна с диаграммой, при этом в качестве вырезанного прямоугольника ему передается область, которую занимало закрытое к этому моменту окно (на рисунке закрашена черным цветом). Теперь необходимо заново нарисовать только эту часть окна с диаграммой; более того, если метод `OnPaint()` и попытается осуществить графический вывод в какой-либо участок рисунка, отличный от того, который был заслонен другим окном, он не сумеет этого сделать. Любые графические операции, выполняемые им, будут игнорироваться. У окна с диаграммой имеется информация об ограничивающем прямоугольнике той части диаграммы, которая относится к автомобилям, и соответственно есть возможность определить, пересекается ли этот прямоугольник с той частью окна, которая была заслонена другим окном. Убедившись в том, что эти две области не пересекаются, программа рисования не будет выполнять повторное рисование той части экрана, которая имеет отношение к автомобилям.

Иногда, когда в процессе рисования возникает необходимость нарисовать только часть какой-либо фигуры или изображения, зачастую оказывается проще нарисовать всю фигуру целиком, а затем вырезать ту часть, которую мы хотим увидеть. Если есть некий образ, то может возникнуть необходимость вывести на экран лишь некоторую его часть. Вместо того чтобы создавать новый образ, который

будет представлять собой требуемую часть исходного образа, можно просто определить вырезаемый прямоугольник, а затем нарисовать этот образ таким образом, чтобы только интересующая нас часть этого рисунка была видна в вырезанной области. Этот способ следует использовать при создании бегущей строки. Последовательно изменяя место, в котором выводится текст, и одновременно задавая вырезаемый прямоугольник, можно получить эффект движущегося в горизонтальном направлении текста.

Наконец, существует способ, который позволяет создавать "окно просмотра" рисунка, имеющего большие размеры. Пользователю предоставляется возможность перемещать это окно по всему рисунку с помощью мыши или используя линейки прокрутки. Окно просмотра может также перемещаться программным путем в зависимости от каких-либо других действий, выполняемых пользователем. В этом случае задание вырезаемого прямоугольника и вывод в него только того, что он позволяет увидеть, является хорошим приемом.

Дополнительную информацию вы можете почерпнуть в свойстве `Clip` класса `Graphics`.

Вывод на печать

GDI+ включает в себя полную поддержку вывода на печать, содержащуюся в классах, которые находятся в пространстве имен `System.Drawing.Printing`. Написание программ, предназначенных для вывода графической информации на принтер, аналогично написанию программ, предназначенных для вывода графической информации на экран. Для этого используются те же самые объекты — `Graphics`, `Point`, `Size`, `Rectangle`, `Pen`, `Brush`, `Image` и т. д., хотя механизм их использования несколько отличается. Вместо того чтобы реагировать на событие `OnPaint()`, мы создаем обработчик событий (который мы можем назвать произвольным образом) и регистрируем его как обработчик событий для вывода страницы:

```
PrintDocument pd = new PrintDocument ();
                   // Предполагается использование принтера по умолчанию
pd.PrintPage += new PrintPageEventHandler(this. pd_PrintPage);
```

Зарегистрировав обработчик, вызывем метод `Print()` класса `PrintDocument`:

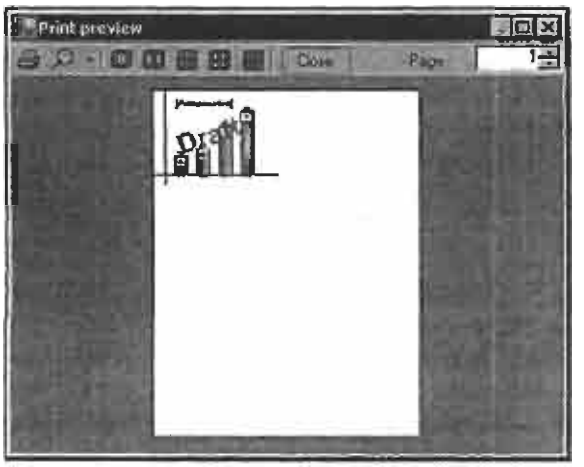
```
pd.Print();
```

Класс `PrintDocument` будет осуществлять повторяющиеся вызовы этого обработчика событий. В тот момент, когда вызывается наступление события, требуемая страница выводится на принтер. Механизм вывода на принтер на самом деле несколько более сложен, но эта схема позволяет получить общее представление о том, каким образом это происходит.

Необходимо принимать во внимание и некоторые другие моменты. У принтера разрешение в пикселях отличается от разрешения экрана, и при выводе графической информации на принтер нужно это учитывать. Некоторые принтеры поддерживают цветную печать, а некоторые — нет. Существует возможность создавать гибкий код, который принимает во внимание все эти моменты и ведет себя достаточно интеллектуально в любых ситуациях.

Предварительный просмотр

Существует возможность создавать окно предварительного просмотра, в котором будет демонстрироваться, каким образом выглядит страница, предназначенная для вывода на печать. Для предварительного просмотра используется тот же самый



обработчик событий, который был написан для вывода на печать. Пользователь, открыв окно предварительного просмотра, получает возможность постраничного просмотра всего документа. Кроме того, есть еще возможность увеличивать и уменьшать масштаб любой части просматриваемой страницы для более подробного ее рассмотрения. Изображение экрана, на который выведено окно предварительного просмотра, приводится на рисунке слева.

Повторим еще раз, что это всего лишь краткий обзор темы, которая рассматривалась в главе 15.

Пространство имен System.Drawing.Drawing2D

Классы, входящие в состав этого пространства имен, позволяют использовать усложненные функциональные возможности для вывода двумерной и векторной графики. Эти классы могут быть использованы для создания сложных приложений, предназначенных для обработки и вывода изображений.

Векторная графика представляет собой способ, при котором программисту вообще не приходится указывать конкретные пиксели. Вместо этого он задает множественные "векторы", среди которых, например, такие операции, как "проведи линию от одной точки до другой", "нарисуй прямоугольник в указанных координатах" и т. д. После это у разработчика программы имеется возможность выполнить масштабирование, вращение и другие преобразования. После осуществления всех необходимых преобразований их результат сразу представляется в соответствующем окне.

В состав пространства имен входят усложненные кисти. Кроме таких кистей, как `LinearGradientBrush` и `PathGradientBrush`, существует еще кисть `HatchBrush`, позволяющая использовать штриховку для рисования и фона, и основного рисунка.

В это же пространство имен входит класс `Matrix`, в котором описаны геометрические преобразования. С помощью этого класса можно видоизменять графические преобразования. Например, с помощью класса `Matrix` можно вывести овал в косой проекции.

В состав этого пространства имен входит также и класс `GraphicsPath`, предоставляющий возможность сначала определить сложный путь, а затем нарисовать его целиком за один присест.

Пространство имен System.Drawing.Imaging

Классы, включенные в это пространство имен, обеспечивают поддержку более сложных возможностей при работе с образами. Две основные группы функциональных возможностей пространства имен — это классы для работы с метафайлами и классы кодирования/декодирования.

В метафайле описывается последовательность осуществляемых графических операций, которую можно сохранить и позднее отыграть назад. Классы `Encoder` и `Decoder` позволяют расширять GDI+, с тем чтобы иметь возможность поддерживать любой произвольный графический формат. Подробнее ознакомиться с ним можно, обратившись к вхождению `System.Drawing.Imaging` в .NET Framework Reference (руководство по .NET Framework).

Итоги

В этой главе были рассмотрены некоторые из классов, входящих в состав пространства имен `System.Drawing`. Мы познакомились с тем, каким образом в класс `Graphics` инкапсулируется поверхность рисования. Мы рассмотрели также механизм рисования, при использовании которого событие `OnPaint()` вызывается каждый раз, когда рисуиок, находящийся в окне, должен быть изменен.

Мы изучили выбор цвета и различные системы координат. Мы познакомились также со структурами `Point`, `Size` и `Rectangle`, которые используются для определения местоположения и размеров поверхности рисования. После этого мы разобрали некоторые примеры, посвященные рисованию линий, фигур, текста и образов.

Перечень тем, рассмотренных в настоящей главе:

- Класс `Graphics`
- Структура `Color`
- Рисование линий с помощью класса `Pen`
- Рисование фигур с помощью класса `Brush`
- Рисование текста с помощью класса `Event`
- Рисование образов с помощью класса `Bitmap`
- Создание образов (двойная буферизация)

Мы узнали, что очень важно вызывать метод `Dispose()` по окончании работы с определенными классами. Это справедливо для следующих классов:

- `Graphics`
- `Pen`
- `Brush`
- `Font`
- `Image`

Наконец, мы представили обзор дополнительных графических возможностей, которыми обладает .NET Framework, а также узнали, к каким разделам документации .NET Framework Reference (руководство по .NET Framework) следует обращаться для получения более детальной информации относительно названных возможностей.

17

Установка приложений Windows

Одной из особенностей .NET является возможность осуществления инсталляции с помощью обыкновенной команды `xcopy`. Модули — это всего лишь некоторый набор файлов, и нет необходимости хранить информацию о конфигурации модулей в реестре, из чего следует, что для инсталляции достаточно просто переписать несколько файлов.

Ниже будет рассмотрен ряд причин, по которым не рекомендуется использовать команду `xcopy` для инсталляции приложений Windows. Для их установки `xcopy` может использоваться только в случае самых простых приложений. Для небольших приложений, инсталлированных лишь на некоторых системах, команда `xcopy` может оказаться вполне пригодной, однако для сложных приложений или приложений, инсталлированных на большом количестве систем, приходится задумываться о другом, более подходящем механизме инсталляции. Команда `xcopy` не позволяет регистрировать модули и отслеживать их местоположение; кроме того, она не может воспользоваться возможностями Windows Installer Zero Administration Windows (ZAW, инсталлятор Windows нулевого уровня администрирования), и, следовательно, какие-либо файлы могут быть непреднамеренно затерты, а также отсутствует встроенная процедура для демонтажа приложения.

> *Инициатива компании Microsoft по созданию средств администрирования нулевого уровня направлена на предоставление системным администраторам возможности автоматически обновлять и инсталлировать приложения на клиентских системах, а также осуществлять централизованное администрирование приложений.*

Система Windows Installer обладает следующими преимуществами для установки приложений Windows:

- В процессе инсталляции пользователь может **выбирать характеристики**, которые должны быть инсталлированы. Существует также возможность последующей инсталляции дополнительных характеристик с помощью **заявки** (advertisement) — в тот момент, когда возникает необходимость в этих характеристиках.
- Если процесс инсталляции терпит неудачу, то происходит **возврат** (rollback), т. е. система возвращается в состояние, в котором она находилась на момент начала инсталляции.
- Если какие-либо файлы оказались испорченными, то легко провести **восстановление** приложения (которое предполагает повторную инсталляцию только испорченных компонентов).

- В специальном файле системы Windows Installer можно найти исчерпывающую информацию о том, какие файлы подлежат инсталляции и какие изменения будут внесены в конфигурацию реестра.
- Есть возможность **полного демонтажа** приложений.

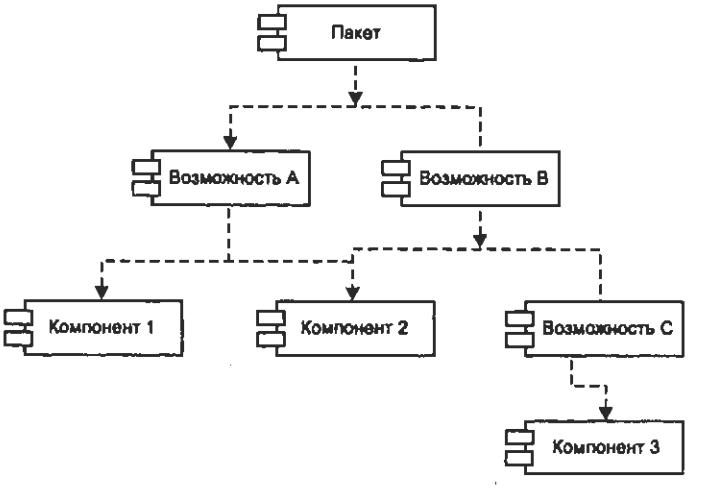
В этой главе мы будем рассматривать:

- Типы устанавливаемых в Visual Studio.NET проектов
- Возможности Windows Installer
- Создание Windows Installer Packages (пакеты инсталлятора Windows) с помощью Visual Studio.NET

Что такое установка?

Установка — это процесс инсталляции приложений на системах, для которых они предназначаются.

Модель обработки (Process Model) системы для работы с решениями компании Microsoft (Microsoft Solutions Framework), к которой можно обратиться по адресу `http://www.microsoft.com/business/services/mcsmsf.asp`, определяет модель обработки как жизненный цикл, состоящий из четырех этапов. Первый этап — это **получение общего представления** о приложении, когда четко формулируется основная идея создаваемого решения. Второй этап — **этап планирования** — состоит из анализа и проработки решения. Третий этап — **этап разработки** включает в себя выполнение основной части разработки. Четвертый этап — **этап стабилизации**, на котором не производится добавление каких бы то ни было новых возможностей, а осуществляется окончательный поиск ошибок и на сайт заказчика отправляется сначала бета-версия, а затем и окончательный вариант системы (см. рис. слева).

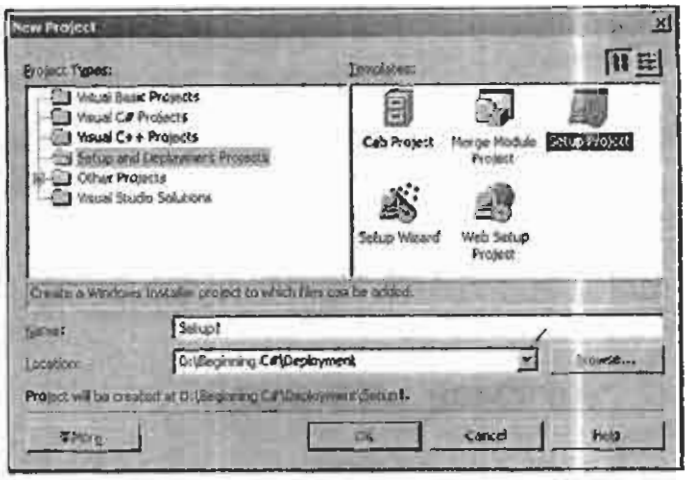


Между этими этапами лежат четыре принципиальных рубежа. Эти рубежи не означают завершение фаз разработки, поскольку в процессе разработки приложения поставленные заказчиком цели могут измениться, а на более поздних этапах может выясниться, что некоторые части системы работают не так, как предполагалось. Поэтому этап планирования может предполагать разработку рабочих прототипов, а на стадии разработки могут уточняться изначальные представления о системе.

При правильном подходе необходимо думать об установке проекта уже на ранних этапах жизненного цикла, поскольку возможные стратегии установки могут оказывать влияние на саму разработку приложения.

Типы проектов по установке

Открыв диалог Add New Project в Visual Studio.NET, можно перейти в следующее диалоговое окно, выбрав пункт меню Setup and Deployment Projects (см. рис. справа).



В этом окне представлены типы проектов и указано, какие действия над ними могут совершаться:

- С помощью шаблона Cab Project (кабинетный проект) могут создаваться кабинетные файлы, предназначенные для объединения нескольких модулей в один файл с последующим его сжатием. Вследствие этого сжатия web-клиент загружает с сервера файлы меньшего объема.

*Создание компонентов не является предметом рассмотрения данной книги, поэтому мы не будем изучать создание кабинетных проектов. Информацию о создании .NET-компонентов, подлежащих загрузке с сервера, можно получить из книги "C# для профессионалов" (**Professional C#**), "ЛОРИ", 2003.*

- Шаблон Merge Module Project (проект по объединению модулей) используется для создания объединенных модулей для Windows Installer. Объединенный модуль — это файл, предназначенный для инсталляции, который может быть включен в мультипрограммные пакеты инсталляции для инсталляторов компании Microsoft. Объединенный модуль может создаваться для компонентов, инсталляция которых требует использования более одной программы инсталляции, после чего такой модуль может включаться в пакеты инсталляции. Одним из примеров объединенного модуля может служить сама система выполнения программ .NET: она поставляется в виде объединенного модуля и, следовательно, может быть включена в пакет инсталляции приложения. Мы воспользуемся объединенным модулем в приложении, которое будет приведено в качестве образца.
- Мы также воспользуемся шаблоном Setup Project (начальная установка проекта). Этот шаблон применяется для создания Windows Installer Package (пакет инсталлятора Windows), и является наилучшим способом установки приложений Windows.
- Setup Wizard (мастер начальной установки) — программа, позволяющая последовательно выбирать необходимые шаблоны. Необходимо ответить на вопрос, что для вас является более предпочтительным: создание программы начальной установки, которая позволяет инсталлировать приложение, или

распространяемого пакета? В зависимости от выбранной возможности будет создан либо пакет для Windows Installer, либо объединенный модуль, либо CAB-файл.

- Последним в данном перечне стоит шаблон Web Setup Project (проект начальной установки web-приложений). В настоящей главе данный шаблон использоваться не будет, поскольку эта книга посвящена созданию приложений Windows. Об установке web-приложений можно прочитать в книге Beginning ASP.NET with C# (ASP.NET и C# для начинающих), выпущенную издательством Wrox (ISBN 1861006152).

Архитектура Windows Installer компании Microsoft

До появления Windows Installer программистам приходилось создавать свои собственные программы, предназначенные для инсталляции приложений. Создание таких программ не только требовало выполнения очень больших объемов работы, но и приводило к тому, что не соблюдались правила, принятые в Windows. Очень часто системные DLL-библиотеки заменялись на более старые версии, поскольку программы инсталляции не производили проверку номера версии. Кроме того, директория, в которую переносились файлы приложения, зачастую оказывалась неправильной, например, использовалась "зашитая" в программу строка `C:\Program Files`. Если системный администратор изменял жесткий диск, используемый по умолчанию, или применял международную версию операционной системы, то инсталляция оканчивалась неудачей.

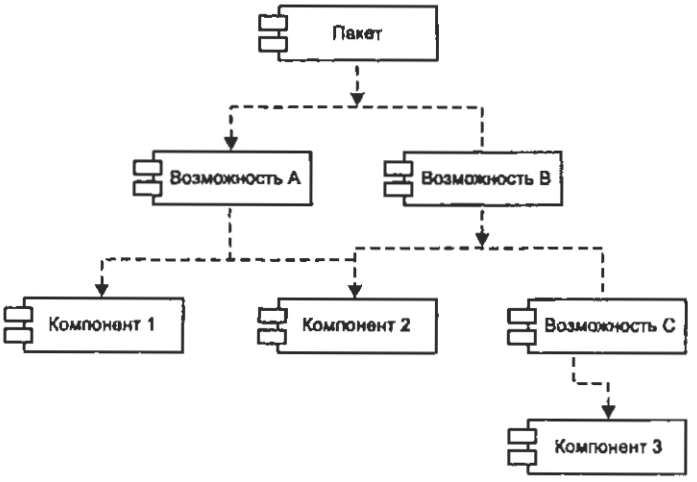
Термины Windows Installer

При работе с Windows Installer нам встретятся некоторые термины, которые применяются в технологии Windows Installer, а именно: пакеты, возможности и компоненты.

Обратите внимание на одно очень важное замечание, касающееся термина "компонент", который используется для программ-инсталляторов:

В контексте Windows Installer компонент представляет собой совсем не то, что подразумевается под этим термином при использовании его в .NET Framework. С точки зрения Windows Installer компонент — это отдельный файл (или несколько логически связанных между собой файлов). Такой файл может быть выполняемым файлом, DLL-файлом или обычным текстовым файлом.

Как можно видеть на следующем рисунке, **пакет** состоит из одной или нескольких возможностей. Пакет — это отдельная база данных Microsoft Installer (MSI). **Возможность** — это некоторая функция продукта с точки зрения пользователя, которая, в свою очередь, может состоять из неких возможностей и компонентов. **Компонент** — это взгляд на процесс инсталляции с точки зрения разработчика; он представляет собой минимальную единицу инсталляции и может состоять из одного или нескольких файлов. Понятия возможность и компонент отличаются друг от друга, поскольку один и тот же компонент может использоваться для различных

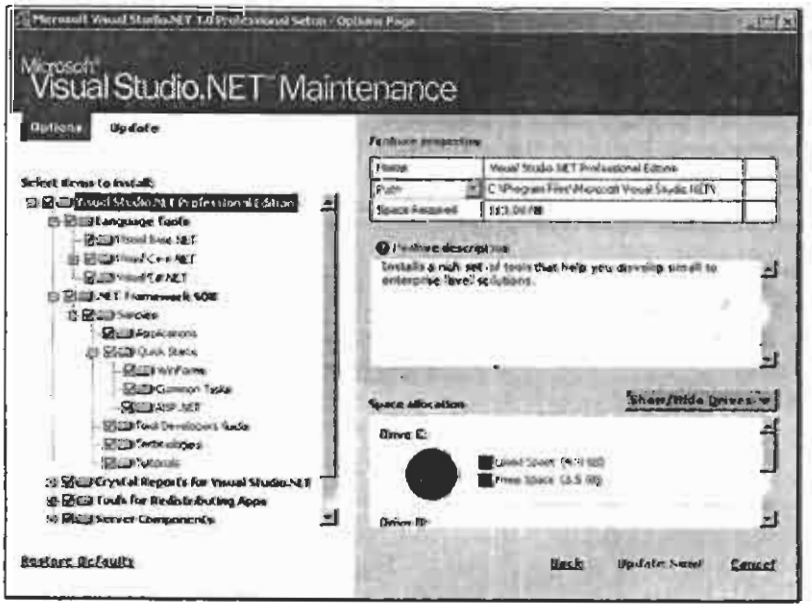


возможностей, как, например, изображенный на рисунке слева Компонент 2. А вот никакая возможность не может входить в состав нескольких других возможностей.

Рассмотрим возможности на примере из реальной жизни, приводимом ранее в данной книге,— Visual Studio.NET. Используя опцию Add/Remove Programs, предоставляемую в Control Panel, можно изменить инсталлированные возможности Visual Studio.NET после окончания ее инсталляции посредством нажатия кнопки Change/Remove.

Нажатие названной кнопки приведет к открытию мастера сопровождения Visual Studio.NET. Обращение же к указанному мастеру — это хороший способ познакомиться с понятием возможности на практике. Как видно на рисунке, приведенном ниже, представленный здесь пакет Visual Studio.NET включает в себя следующие возможности: Language Tools (инструменты для работы с различными языками программирования), .NET Framework SDK, Crystal Reports for Visual Studio.NET, Tools for Redistributing Apps (инструменты для вторичного распространения приложений), Server Components (серверные компоненты). Возможность Language Tools, в свою очередь, включает в себя следующие возможности более низкого уровня: Visual C++.NET, Visual Basic.NET, Visual C#.NET и т. д.





Преимущества Windows Installer

Windows Installer обладает следующими преимуществами:

- Каждая возможность может быть инсталлирована, не быть инсталлирована или быть объявлена. **Объявленная** возможность, входящая в состав пакета, будет инсталлирована сразу, как только возникнет такая необходимость. Возможно, вы уже сталкивались с такой ситуацией, когда Windows Installer запускался в процессе вашей работы с Microsoft Word. Как только вы попытаетесь воспользоваться какой-либо объявленной, но не инсталлированной возможностью, то в этот момент будет осуществлена автоматическая инсталляция требуемой возможности.
- Если приложение окажется испорченным, то в этом случае оно может **самовосстановиться** с помощью возможностей восстановления, предусмотренных в пакетах Windows Installer.
- Если процедура инсталляции завершится неудачей, будет автоматически выполнен откат. После неудачной попытки инсталляции все будет возвращено в свое первоначальное состояние: в системе не останется никаких дополнительных ключей реестра, никаких файлов и т. п.
- С помощью процесса **обратной инсталляции** все файлы, все ключи реестра и т. д. будут удалены, т. е. существует возможность полной обратной инсталляции приложения. При этом не останется никаких временных файлов, а состояние реестра будет возвращено в исходное состояние.

Из таблиц базы данных MSI можно получить полную информацию о том, какие файлы скопированы, какие ключи занесены в реестр и т. д.

Создание пакета инсталляции для приложения Simple Editor

Для создания пакета Windows Installer с помощью Visual Studio.NET мы используем решение, которое было создано в главе 15,— Simple Editor. Для этих целей можно воспользоваться любыми приложениями Windows Form, которые создавались в процессе выполнения практикума; в этом случае вам придется лишь изменить имена соответствующим образом.

Планирование инсталляции

Прежде приступить к построению программы инсталляции, необходимо определиться, что именно мы планируем включить в нее. Существует несколько вопросов, на которые нужно ответить в первую очередь:

- **Какие файлы необходимы для данного приложения?** Естественно, все выполняемые файлы и, возможно, некоторые модули компонентов. Нет необходимости определять все файлы, от которых они зависят,— эти файлы будут включены автоматически. Возможно, возникнет необходимость включить и какие-либо другие файлы. Это может относиться к файлу с документацией,

readme.txt, файлу с лицензией, шаблону документа, графическим файлам, файлам конфигурации и т. д. Необходимо учитывать все файлы, которые подлежат включению.

Для конкретного приложения Simple Editor нам потребуется выполняемый файл, и, кроме того, мы включим файлы readme.rtf, license.rtf и двоичный файл с изображением логотипа Wrox Press, который будет выводиться инсталляционными диалогами.

- **Какие директории должны использоваться?** Файлы приложения должны инсталлироваться в директории Program Files\Application name. Директория Program Files имеет различное название у различных версий операционных систем, предназначенных для эксплуатации в разных странах. Кроме того, администратор может выбрать иной путь для данного приложения. Нет необходимости знать, где эта директория располагается на самом деле, поскольку существует возможность обратиться к соответствующей функции API, которая вернет интересующую нас директорию. С помощью инсталлятора можно использовать Эспециальную заранее определенную папку для размещения файлов в директории Program Files.

Ни при каких обстоятельствах не следует "зашивать" имена директорий в тело программы. В различных международных версиях эти директории имеют различные имена! Даже если ваше приложение рассчитано на работу исключительно с англоязычными версиями Windows (чего также на самом деле делать не следует), существует вероятность того, что системный администратор мог переопределить местоположение этих директорий.

Приложение Simple Editor будет хранить выполняемый файл в директории, используемой приложением по умолчанию, если только пользователь, выполняющий инсталляцию, не выберет какой-либо другой путь.

- **Каким образом пользователь будет осуществлять доступ к данному приложению?** Существует возможность вызывать выполняемый файл, включив ссылку на него в меню Start, разместив иконку на рабочем столе и т. д. Если вы собираетесь размещать иконку на рабочем столе пользователя, то необходимо выяснить, устраивает ли это самого пользователя. Общее руководство по Windows XP рекомендует по возможности не включать в рабочий стол ничего лишнего.

  Приложение Simple Editor будет вызываться через меню Start.

- **На каком носителе будет распространяться данное приложение?** Хотим ли мы, чтобы пакеты инсталляции записывались на CD, на дискету или были общедоступны из сети?

- **Что мы будем запрашивать у пользователя?** Должен ли он подтвердить свое согласие с лицензионной информацией, следует ли выводить файл Readme, требуется ли запрашивать путь инсталляции? Существуют ли какие-либо опции при инсталляции?

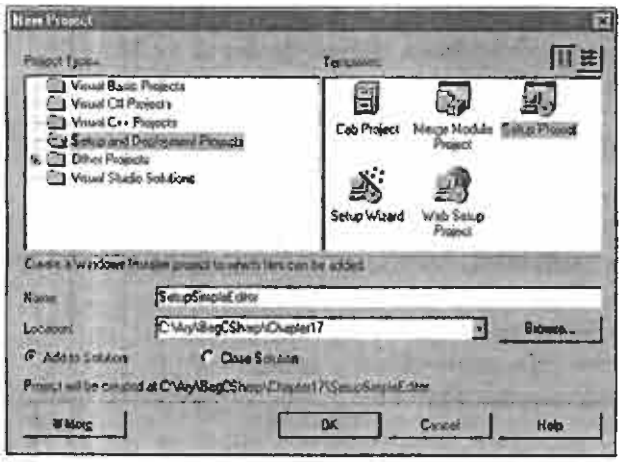
Для данного примера будут использоваться диалоговые окна с инсталлятором Visual Audio.NET. Нам предложат указать директорию, где следует инсталлировать

программу (можно задать другой путь, отличный от заданного по умолчанию), далее будет показан справочный файл `Readme`, а затем будет предложено ввести информацию лицензионного характера.

Создание проекта

Теперь можно воспользоваться инсталлятором Visual Studio.NET для создания проекта по инсталляции и включить в него все файлы, подлежащие инсталляции. Первыми шагами будут использование мастера Project (проект) и конфигурирование проекта.

Практикум: создание проекта Windows Installer



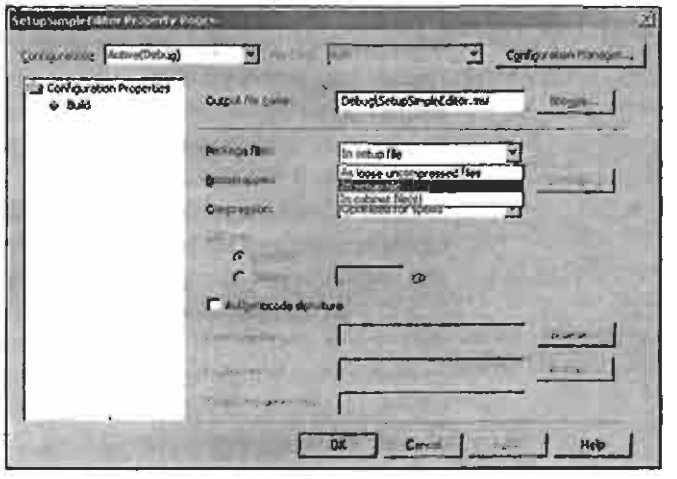
1. Откройте файл с решением приложения Simple Editor, созданного при изучении главы 15. Мы добавим инсталляционный проект к уже существующему решению. Если это решение из главы 15 вами не создавалось, то вы можете загрузить его с web-сайта издательства Wrox, расположеиного по адресу `http://www.wrox.com`.
2. Добавьте Setup Project к решению с помощью пункта меню **File | New | Project**. Выберите переключатель **Add to Solution** и нажмите кнопку **OK** (см. рис. слева).

Свойства проекта

На настоящий момент в проекте есть только файл для начальных установок решения. Теперь необходимо определить файлы, подлежащие инсталляции. Кроме того, в любом случае будет необходимо сконфигурировать свойства проекта. Для этого следует познакомиться с опциями **Packaging** (упаковка) и **Bootstrapper** (самоинсталляция).

Опция Packaging

MSI — это база данных, в которой начинается процесс инсталляции, однако имеется возможность определить, каким образом упакованы предназначенные для инсталляции файлы, для чего можно воспользоваться тремя опциями в диалоговом окне, представленном на рисунке справа. Этот диалог откроется после выбора вами пункта **Project | Properties** в меню **Visual Studio.NET**. Ниже подробно описаны три упомянутые опции.



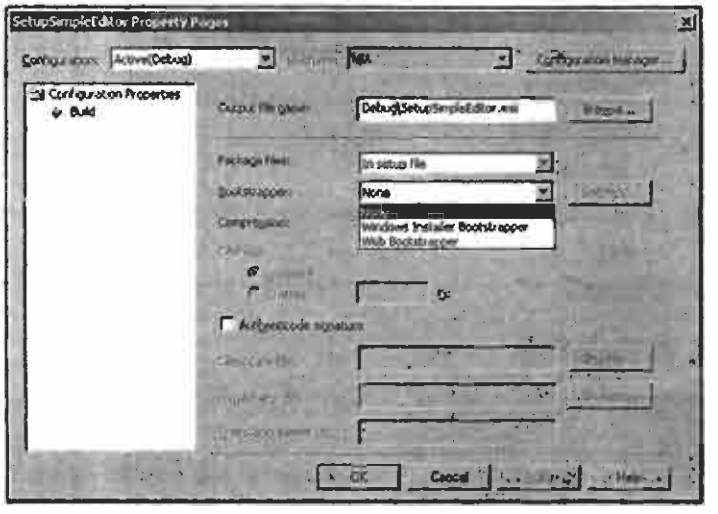
- В режиме **loose uncompressed files** (в виде несвязанных несжатых файлов) все программные файлы и файлы с данными будут храниться в своем первозданном виде. Никакого сжатия файлов при этом не производится.
- Режим **In Setup file** (в файле начальной установки) означает, что все необходимые файлы будут объединяться и сжиматься в MSI-файл. Для отдельных компонентов в пакете значение этой опции может быть переопределено. Если вы собираетесь поместить все свои файлы в единственный MSI-файл, то следует обратить внимание на то, поместится ли программа инсталляции на предназначенном для нее носителе, например, на CD или на дискете. Если объем предназначенных для инсталляции файлов настолько велик, что емкости дискеты оказывается недостаточно, можно попытаться изменить степень сжатия ваших файлов, воспользовавшись опцией **Optimized for Size** (оптимизация с учетом объема) в окне со списком **Compression** (сжатие). Если объема все равно оказывается недостаточно, то можно воспользоваться следующей опцией для упаковки.
- Третьим способом упаковки файлов является использование **кабинетных файлов** (CAB-файлов). В этом случае MSI-файл используется только для того, чтобы загружать и инсталлировать CAB-файлы. При использовании CAB-файлов можно задавать их размеры, которые позволят использовать для инсталляции компакт-диски или дискеты (при инсталляции с дискет следует установить размер, равный 1440 Кб).

Программа самоинсталляции Bootstrapper

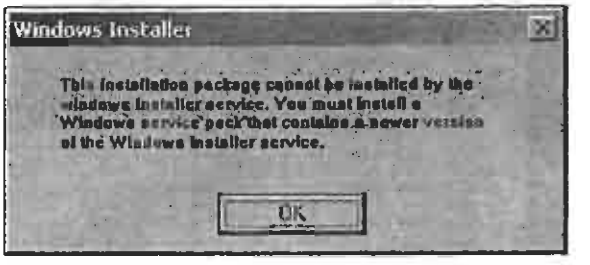
В этом же диалоге можно сконфигурировать еще одну опцию — **программу самоинсталляции**, которая должна быть выполнена до того, как появится возможность запускать собственно инсталлируемое приложение.

В системе, где производится инсталляция, необходимо наличие Windows Installer версии 1.5 для пакетов инсталляции, созданных с помощью Visual Studio.NET. Windows Installer версии 1.5 начал впервые поставляться с Windows XP. Если инсталлируемую программу требуется установить на системах более ранних версий, то необходимо использовать **программу самоинсталляции**, которая осуществляет инсталляцию последней версии Windows Installer. Выбрав опцию **Windows Installer Bootstrapper** (программа самоинсталляции Windows Installer), вы включите программу самоинсталляции в общую программу инсталляции, что при осуществлении инсталляции приведет сначала к установке Windows Installer версии 1.5, и только после этого начнется инсталляция требуемой программы. Для этой опции необходимо около 3 Мбайт.

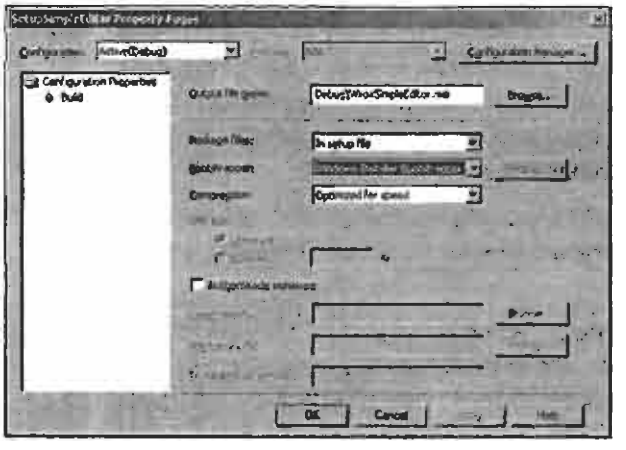
При использовании опции **Web Bootstrapper** требуется разместить программу самоинсталляции на web-сайте. Когда в Properties Editor выбирается данная опция, происходит запрос URL, с которого должна происходить загрузка. Этот URL будет в дальнейшем использоваться программистом, осуществляющим инсталляцию приложения, для установки программы самоинсталляции. При использовании данной опции не требуется дополнительного пространства для пакета инсталляции, но пользователь, осуществляющий инсталляцию, должен обязательно обладать доступом к Интернету.



Если же вы выберите опцию None (см. рис. слева), которая не предполагает установку программы самоинсталляции, а на системе, на которой будет производиться инсталляция, Windows Installer версии 1.5 отсутствует, то в процессе инсталляции будет получено сообщение: "This installation package cannot be installed by the Windows Installer service. You must install a Windows service pack that contains a newer version of the Windows Installer service." ("Данный пакет инсталляции не может быть инсталлирован службой инсталляции Windows. Вам необходимо инсталлировать сервисный пакет Windows, в котором содержится более поздняя версия службы Windows Installer.") (см. рис. справа).



Практикум: конфигурирование проекта



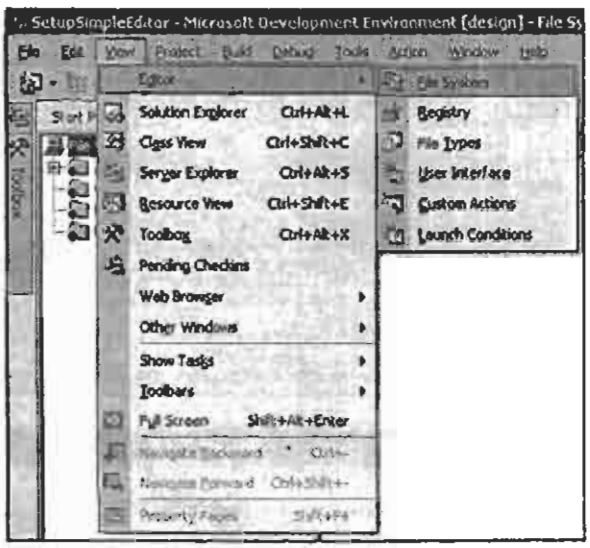
1. Выберите для опции Bootstrapper на страничке определения свойств (View | Property Pages) режим Windows Installer Bootstrapper, что сделает возможной инсталляцию приложения на системах, в которых отсутствует Windows Installer версии 1.5. Кроме того, измените имя выходного файла на `WroxSimpleEditor.msi`, как показано на рисунке слева.
2. Установите свойства проекта в соответствии со значениями, приведенными в таблице:

Свойство	Значение
`Author` (автор)	Кристиан Нейджел
`Description` (описание)	Simple Editor — простой редактор для редактирования и вывода на печать текстовых файлов.
`Keywords` (ключевые слова)	Installer (инсталлятор), Wrox Press, Simple Editor
`Manufacturer` (изготовитель)	Wrox Press
`ManufacturerUrl` (URL изготовителя)	`http://www.wrox.com`

Продолжение таблицы

Свойство	Значение
`ProductName` (наименование продукта)	Simple Editor
`SupportUrl` (URL сопровождения)	`http://p2p.wrox.com`
`Title` (заголовок)	Инсталляционная база данных для Simple Editor
`Version` (версия)	1.0.0

Редакторы начальных установок

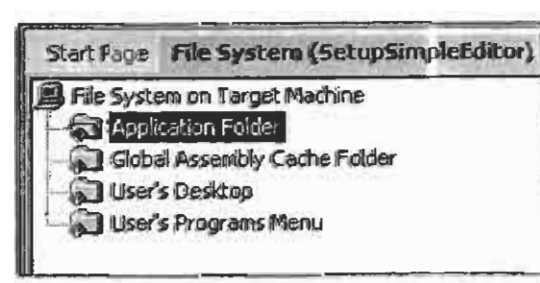


В Visual Studio.NET Setup Project есть шесть различных редакторов. Можете воспользоваться одним из них, открыв проект по осуществлению установки и выбрав меню View | Editor, как показано на рисунке слева.

- Редактор **File System** (файловой системы) используется для включения файлов в пакет инсталляции.
- Редактор **Registry** (реестра) позволяет создавать ключи реестра для данного приложения.
- Редактор **File Types** (типов файлов) позволяет регистрировать специфические для данного приложения расширения файлов.
- Редактор **User Interface** (пользовательского интерфейса) позволяет включать новые и конфигурировать уже существующие диалоги, выводящиеся при инсталляции продукта.
- Редактор **Custom Actions** (нестандартных действий) позволяет запускать нестандартные программы в процессе осуществления инсталляции или обратной инсталляции.
- Редактор **Launch Conditions** (условий запуска) позволяет задавать требования, необходимые для инсталлируемых приложений, например, требование постоянного присутствия .NET Runtime.

Редактор файловой системы

Редактор файловой системы предназначен для включения файлов в пакет инсталляции и для задания местоположения, в котором они должны инсталлироваться. Этот редактор открывается с помощью меню View | Editor | File System в Visual Studio.NET. При этом открываются некоторые предварительно описанные специальные папки, как показано на рисунке справа. Ниже приводится описание назначения каждой папки.



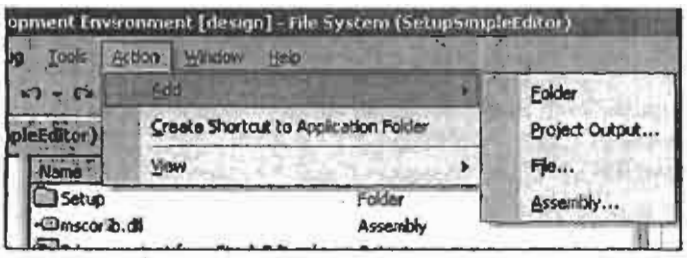
- **Application Folder** (папка приложений) используется для хранения выполняемых файлов и библиотек. Ее местоположение определяется как `[Progra,FilesFolde] [Manufacture] [ProductName]` (*`[ПапкаПрограммныхФайлов] [Производитель] [НаименованиеПродукта]`*). В англоязычных системах *`[ПапкаПрограммныхФайлов]`* заменяется на `C:\Program Files`. Директории для *`[Производитель]`* и *`[НаименованиеПродукта]`* определяются свойствами проекта `Manufacturer` и `ProductName`.
- В **Global Assembly Cash Folder** (папка для кэшированных глобальных модулей) могут инсталлироваться модули общего доступа. Эта папка используется для модулей, которые будут использоваться несколькими приложениями. С модулями общего доступа вы сможете познакомиться более детально в главе 21, а еще более подробную информацию можно найти в книге "C# для профессионалов" (Professional C#), "ЛОРИ", 2003.
- Если требуется поместить иконку на рабочем столе, то следует воспользоваться опцией **User Desktop**. По умолчанию к этой папке ведет один из следующих путей — `C:\Documents and Settings\` *`[имя пользователя]`*`\Desktop` или `C:\Documents and Settings\All Users\Desktop` — в зависимости от того, осуществляется ли инсталляция для отдельного пользователя или для всех пользователей.
- Пользователь обычно запускает программу из **User Programs Menu** (меню пользовательских программ). В качестве пути по умолчанию используется `C:\Documents and Settings\[`*`Имя пользователя`*`]\Start Menu\[`*`Программы`*`]`. Можно включить в это меню пункт для упрощенного вызова приложения, который следует включить названия компании-пронзводителя и приложения, с тем чтобы пользователю было легко идентифицировать приложение, например, Microsoft Excel.

  Некоторые приложения создают подменю, из которых может запускаться более одного приложения; в качестве примера можно привести Visual Studio.NET 7.0. В соответствии с Windows Guidelines (руководство по Windows) многие программы не имеют причин поступать таким образом. В эти меню не следует включать программу обратной инсталляции, поскольку она доступна из раздела Add/Remove Programs, находящегося в Control Panel, и должна вызываться именно оттуда. В эти меню не следует также включать help-файл, поскольку к нему должен быть доступ непосредственно из самого приложения. Поэтому для большинства приложений будет вполне достаточно строки непосредственного вызова, расположенного в `User Programs Menu`. Цель этих ограничений — не загромождать меню Start слишком большим количеством пунктов.

Подробное изложение этой информации может быть найдено в документе "Application Specification for Microsoft Windows 2000 for desktop applications" (Спецификации для desktop-приложений для Microsoft Windows 2000). Этот документ может быть найден по адресу `http://msdn.microsoft.com`. Воспользуйтесь пунктом меню Partners & Certification | Windows Logo Program.

- **User Personal Data Folder** (папка персональных данных пользователя) по умолчанию выбирается в качестве папки, в которой будут храниться все документы. В качестве пути по умолчанию используется `C:\Documents and Settings\[имя пользователя]\My Documents`. Этот путь по умолчанию используется также и для директории, в которой хранит свои проекты Visual Studio.NET.
- Добавление строки непосредственного вызова в **User Send To Menu** (меню пользователя Send To) приводит к соответствующему расширению контекстного меню Send To (отправка) в момент выбора файла. Это контекстное меню позволяет отправить файл какой-либо программе, например, драйверу дискеты, получателю почты или в папку `My Documents`.

Добавление элементов в специальные папки



Выбрав некоторую папку и воспользовавшись пунктом меню Action | Add, мы получаем возможность добавлять элементы в специальные папки, как показано на рисунке слева.

Можно выбрать из следующих элементов: **Subfolder** (подпапка), **File** (файл), **Assembly** (модуль) и **Project Output** (выходной поток проекта). Направление выходного потока проекта в папку приведет к автоматическому включению в эту папку сгенерированных выходных файлов с расширением .DLL или .EXE в зависимости от того, является ли данный проект библиотекой компонентов или приложением. Выбор любого из пунктов меню Project Output или Assembly приведет к тому, что в папку будут автоматически добавлены все файлы, от которых зависит приложение (все модули, на которые имеются ссылки).

Свойства файла

Для файла, находящегося в папке, могут быть определены следующие свойства. В зависимости от типа конкретного файла некоторые из перечисленных свойств могут оказаться неприменимыми, кроме того, существуют некоторые дополнительные свойства, не вошедшие в приведенную ниже таблицу:

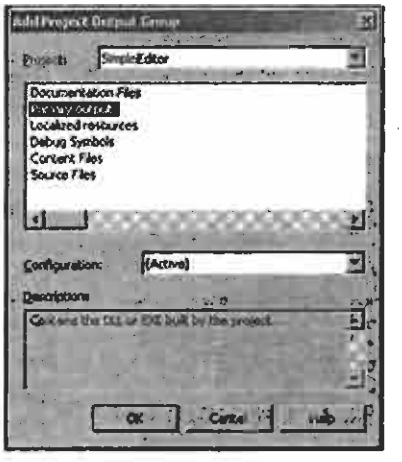
Свойство	Описание
`Condition` (условие)	С помощью этого свойства определяется условие, при котором будет осуществляться инсталляция выбранного файла. Это может оказаться полезным, если этот файл должен использоваться только для определенных версий операционных систем или в зависимости от того, какую информацию задаст пользователь в процессе диалога.
`Exclude` (исключение)	Этому свойству может быть присвоено значение `True`, если данный файл не подлежит инсталляции. Это позволяет файлу входить в состав проекта, но при этом не инсталлироваться. Исключать файлы можно, если вы уверены либо в том, что на них никто не ссылается, либо в том, что такие файлы уже существуют на всех системах, на которых будет инсталлироваться данное приложение.

Продолжение таблицы

Свойство	Описание
`PackageAs` (способ упаковки)	Это свойство позволяет переопределять способ добавления файлов к пакету инсталляции, использующемуся по умолчанию. Например, если в конфигурации проекта используется режим In Setup File, то для какого-либо конкретного файла этот режим может быть изменен на `Loose`, что приведет к тому, что данный файл не будет включаться в файл базы данных MSI. Это может оказаться особенно полезным, когда требуется включить файл `Readme`, который пользователю следует прочесть до начала инсталляции. Совершенно очевидно, что такой файл не следует подвергать сжатию, даже если сжатию подвергаются все остальные файлы.
`Permanent` (постоянный)	Если этому свойству присвоено значение `True`, это означает, что файл останется на компьютере даже после того, как будет выполнена процедура обратной инсталляции продукта. С одним примером такого файла вы могли столкнуться при инсталляции новой версии Microsoft Outlook. Если вы сначала конфигурируете Microsoft Outlook, затем выполняете процедуру обратной инсталляции, после чего инсталлируете его вновь, необходимость конфигурировать его заново отсутствует, поскольку файл конфигурации, оставшийся от предыдущей инсталляции, не подлежит уничтожению.
`ReadOnly` (Только чтение)	Это свойство позволяет задавать файлу при инсталляции атрибут "только чтение".
`Vital` (жизненно важный)	Это свойство означает, что данный файл существен для инсталляции данного продукта. Если инсталляция этого файла завершается неудачно, то вся процедура инсталляции прекращается и выполняется откат.

Практикум: добавление файлов в пакет инсталляции

1. Включите основной выходной поток проекта Simple Editor в проект по созданию инсталлятора с помощью меню Project | Add | Project Output. В диалоге Add Project Output Group выберите Primary Output как показано на рисунке справа.



Нажатие кнопки OK приведет к тому, что основной выходной поток проекта `SimpleEditor` будет добавлен в Application Folder в автоматически открытом редакторе системных файлов. В данном случае основным выходным потоком является `SimpleEditor.exe`. Как можно увидеть в Solution Explorer (View | Solution Explorer), все файлы, от которых он зависит, определены автоматически; это привело к включению в проект объединенного модуля с именем `dotnetfxredist_x86_enu.msm`, в который включены все файлы, необходимые для работы системы выполнения .NET. Если на системе, на которой будет устанавливаться данное приложение, система выполнения .NET отсутствует, то она будет инсталлирована вместе с приложением.

Для нашей программы инсталляции мы налагаем требование предварительной установки системы выполнения .NET. Выберите файл `dotnetfxredist_x86_enu.msm` и измените значение свойства `Exclude` на `True`. Для инсталляции системы выполнения .NET требуется приблизительно 15 Мбайт, поэтому теперь наш файл инсталляции станет меньше на 15 Мбайт. Простое изменение значения

свойства Exclude на True приведет к тому, что процедура инсталляции будет успешно завершаться, даже если .NET не инсталлирована на этой системе; в этом случае SimpleEditor.exe просто не будет запускаться. Лучшим решением этой проблемы было бы выдавать в процессе инсталляции сообщение об ошибке, если система выполнения .NET данном компьютере отсутствует. Этого можно добиться, описав условие запуска в .NET Framework. Мы вернемся к этому чуть позже.

2. Дополнительные файлы, которые мы хотим включить,— это файлы с логотипом и лицензией, а также файл readme. В редакторе файловой системы создайте поддиректорию Application Folder с именем Setup. Это можно сделать, перейдя в Application Folder и выбрав меню Action | Add | Folder.

Меню Action доступно в Visual Studio.NET, только если выбираются элементы в редакторах начальных установок. Если выбран какой-либо элемент в Solution Explorer или Class View, меню Action оказывается недоступным.

3. Перенесите с помощью мыши файлы wroxlogo.bmp, wroxsetuplogo.bmp, readme.rtf и license.rtf в папку Setup. Эти файлы можно найти среди примеров программ для данной книги на http://www.wrox.com, однако их нетрудно создать самостоятельно, для чего достаточно поместить в текстовые файлы информацию о лицензии и информацию, которую следует прочитать пользователю. Нет необходимостн изменять свойства этих файлов. Эти файлы будут использоваться в диалогах программы инсталляции.

4. Перенесите мышью файл readme.txt в папку приложения. Необходимо, чтобы у пользователя имелась возможность прочитать этот файл до начала процедуры инсталляции. Присвойте свойству PackageAs значение vsdpaLoose, для того чтобы этот файл не подвергался сжатию в пакет инсталляции. Присвойте значение True свойству ReadOnly, с тем чтобы в этот файл нельзя было вносить изменения.

5. Перенесите мышью файл demo.wroxtext в папку User's Desktop. Этот файл должен инсталлироваться только после того, как пользователь подтвердит, что установка этого файла ему действительно необходима. С этой целью присвойте свойству Condition значение CHECKBOXDEMO. Это значение обязательно должно быть записано заглавными буквами. Данный файл будет инсталлироваться, только если это условне будет иметь значение true. Несколько позже мы опишем диалог, в котором это свойство будет задаваться.

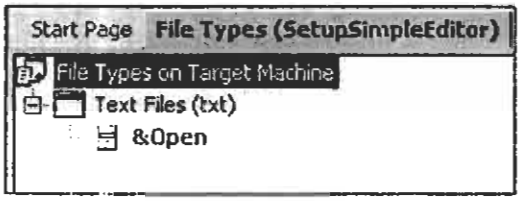
Автору кажется более предпочтительным демонстрировать свойство Condition посредством строк непосредственного вызова, а не с помощью файлов, поскольку в User's Desktop лучше помещать именно их. Однако на момент написания данной книги в инсталляторе Visual Studio.NET содержалась ошибка, в результате которой свойство Condition не могло задаваться для строк непосредственного вызова, хотя в MSDN-документации утверждалось обратное.

Мы бы хотели иметь возможность вызова программы из меню Start | Program. Нам потребуется строка непосредственного вызова для программы `SimpleEditor`.

6. Выберите элемент Primary output from SimpleEditor в `Application Folder` и откройте меню Action | Create Shortcut to Primary output from SimpleEditor. Присвойте свойству `Name` создаваемой строки непосредственного вызова значение `Wrox Simple Editor` и перенесите ее мышью в `User's Programs Menu`.

Редактор типов файлов

Если в приложении используются типы файлов, созданные пользователем, и требуется зарегистрировать расширения этих файлов, которые позволят запускать приложение с помощью двойного щелчка мышью на таком файле или через контекстное меню, то для этого можно использовать редактор типов файлов, который может быть запущен с помощью View | Editor | File Types (см. рис. слева).



С помощью редактора типов файлов можно сконфигурировать расширение файлов, которое будет использоваться приложением. Расширение файла обладает следующими свойствами:

Свойство	Описание
`Name` (имя)	Здесь следует указать имя, которое содержит полезную информацию в смысле описания типа файла. Это имя выводится в редакторе типов файлов, а также заносится в реестр. Имя должно быть уникальным. В качестве примера для типов файлов с расширением `.doc` можно привести `Word.Document.8`. Совершенно необязательно использовать ProgID, как в примере в Word, простого текста вроде `wordhtmlfile`, который используется для расширения файлов `.dochtml`, будет вполне достаточно.
`Command` (команда)	С помощью этого свойства можно задать выполняемый файл, который будет запускаться, когда пользователь открывает некоторый файл данного типа.
`Description` (описание)	Здесь может быть размещено описание.
`Extensions` (расширения)	Расширение файла, с которым ваше приложение должно быть зарегистрировано. Данное расширение файла будет зарегистрировано в разделе реестра.
`Icon` (иконка)	Позволяет задавать иконку, которая будет выводиться для данного расширения файла.

Создание действий

После того как с помощью редактора типов файлов созданы новые типы файлов, можно добавлять действия. Действие, которое автоматически добавляется по умолчанию,— это Open (открыть). Существует возможность добавлять дополнительные действия, например, New (создать новый), Print (печатать) и вообще любые действия, которые программа может осуществлять над файлами. Для каждого действия должны быть определены свойства `Arguments` (аргументы) и `Verb` (глагол). Свойство `Arguments` используется для задания аргумента, который передается программе, зарегистрированной для данного расширения; например, "%1"

означает, что приложению передается имя файла. Свойство `Verb` определяет, какое именно действие должно произойти. Для действия "вывод на печать" может быть добавлено `/print`, если это поддерживается приложением.

Теперь добавим какое-либо действие в программу инсталляции SimpleEditor. Нам требуется зарегистрировать расширение файла, с тем чтобы приложение SimpleEditor могло использоваться из Windows Explorer для открытия файлов с расширением `.wroxtext`. После осуществления подобной регистрации появляется возможность открывать такие файлы с помощью двойного щелчка на них мышью, при этом приложение SimpleEditor будет запускаться автоматически.

Практикум: установка расширения файла

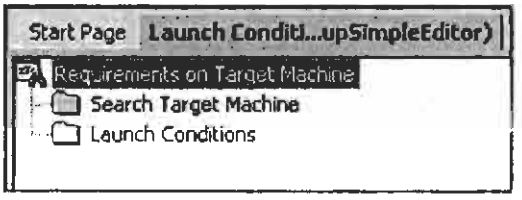
1. Запустите редактор типов файлов через **View | Editor | File Types**. Воспользовавшись меню **Action | Add File Type**, добавьте новый тип файла со свойствами, перечисленными в таблице справа. Мы не собираемся нарушать право собственности Notepad на файлы с расширением `.txt`, поэтому будет использоваться расширение файлов `.wroxtext`.

Свойство	Значение
`(Name)`	`Wrox.SimpleEditor.Text`
`Command`	Основной выходной поток SimpleEditor
`Description`	Текстовые документы Wrox
`Extensions`	`wroxtext`

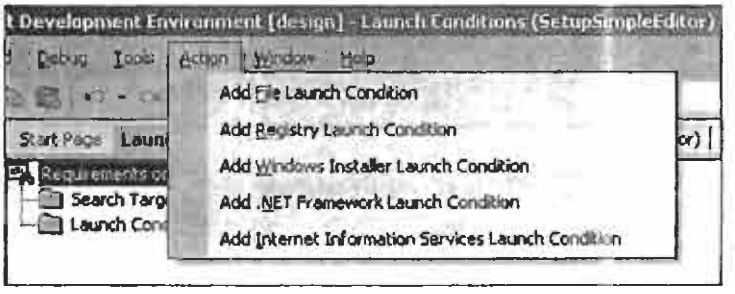
Можно также задать свойство `Icon`, которое будет определять иконку, используемую при открытии файлов.

Сохраните для свойств действия **Open** значения, используемые по умолчанию; в этом случае имя файла будет передаваться приложению в качестве аргумента.

Редактор условий запуска



С помощью **Launch Condition Editor:** (см. рис. слева) можно предъявлять определенные требования к системе, которые должны быть выполнены до начала инсталляции. Редактор условий запуска может быть вызван через меню **View | Editor | Launch Conditions**. В указанном редакторе есть два раздела для задания требований: **Search Target Machine** (поиск требуемого компьютера) и **Launch Conditions** (условия запуска). В первом разделе определяется, что и каким образом следует искать для конкретного файла, ключа реестра и т. д., а во втором разделе задаются сообщения об ошибке, которые будут выведены, когда поиск оказывается безуспешным.



Рассмотрим условия запуска, которые можно определить с помощью меню **Action** (см. рис. справа).

- С помощью **File Launch Condition** можно осуществлять поиск файлов, инсталлированных на том компьютере, на котором будет осуществляться инсталляция.

- **Registry Launch Condition** позволяет осуществлять поиск ключей реестра до начала инсталляции.
- **Windows Installer Launch Condition** позволяет осуществлять поиск файлов Windows Installer.
- **.NET Framework Launch Condition** осуществляет проверку на предмет того, что .NET Framework уже инсталлирована на том компьютере, на котором должна осуществляться инсталляция.
- **Internet Information Services Launch Condition** осуществляет поиск инсталлированного Internet Information Server (информационный сервер Интернета). Есть также возможность проверить версию этого сервера.

В нашем примере мы воспользуемся условием .NET Framework Launch Condition.

.NET Framework Launch Condition

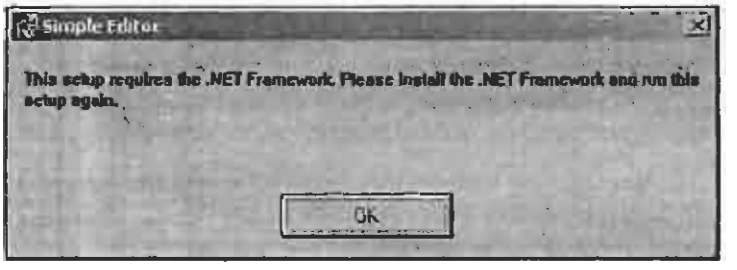
Добавление .NET Framework Launch Condition с помощью редактора условий запуска приводит к добавленню единственного условия с заранее определенными значениями свойств: свойство `Condition` имеет значение `MsiNetAssemblySupport`, а свойство `Message` — `[VSDNETMSG]`. Свойству `MsiNetAssemblySupport` в процессе инсталляции Windows Installer 1.5 присваивает значение `true`, если на данной системе установлена Common Language Runtime (общая система выполнения). Сообщение [VSDNETMSG] — это сообщение, которое будет выводиться по умолчанию, если Common Language Runtime не инсталлирован.

Практикум: добавление .NET Framework Launch Condition

Мы исключили систему выполнения .NET из файлов с помощью редактора системных файлов, следовательно, необходимо осуществить проверку на предмет наличия инсталлированной системы выполнения .NET на системе, на которой будет производиться инсталляция:

1. Если вы еще не запустили редактор условий запуска, то сделайте это через View | Editor | Launch Conditions.
2. С помощью Action | Add .NET Framework Launch Condition добавьте .NET Launch Condition. Свойству `Name` можно присвоить значение по своему усмотрению, например, `DOTNETCondition`.

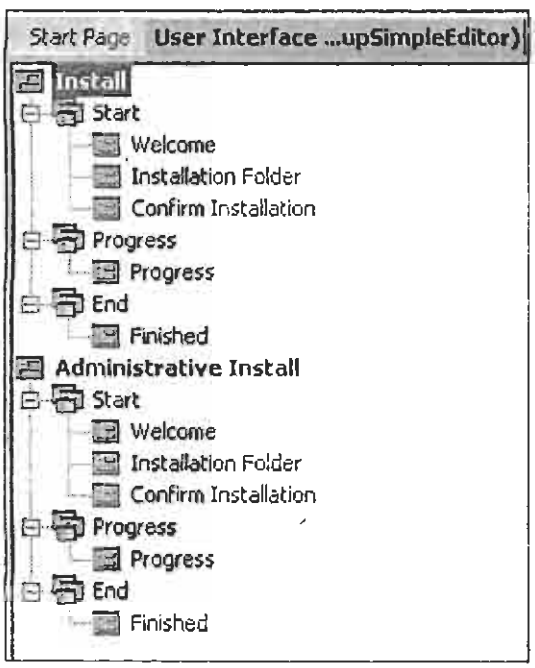
При инсталляции нашего приложения Windows Installer будет осуществлять проверку на предмет наличия системы выполнения .NET. Если окажется, что названная система не инсталлирована, то пользователь получит сообщение об ошибке: "Данная начальная установка требует наличия .NET Framework. Пожалуйста, инсталлируйте .NET Framework и запустите эту программу еще раз." (см. рис. слева), а процедура инсталляции будет прекращена.



Редактор пользовательского интерфейса

Редактор пользовательского интерфейса позволяет определять диалоги, с помощью которых пользователь имеет возможность конфигурировать процедуру инсталляции. В них можно проинформировать пользователя о лицензионных соглашениях, запросить пути инсталляции и другую информацию, необходимую для конфигурирования приложения.

Практикум: запуск редактора пользовательского интерфейса



Запустите редактор пользовательского интерфейса, выбрав View | Editor | User Interface. Редактор пользовательского интерфейса используется с целью задания свойств заранее описанным окнам диалога. Вы должны увидеть на экране автоматически сгенерированные диалоги и два режима инсталляции, как показано на рисунке слева.

Режимы Install и Administrative Install

На рисунке изображены два различных режима инсталляции — Install (инсталляция) и Administrative Install (административная инсталляция). Режим **Install** — это обычная инсталляция, которая используется при необходимости инсталлировать приложение на конкретной системе. Режим **Administrative Install** позволяет инсталлировать общедоступный образ приложения в сети. В этом случае пользователь имеет возможность самостоятельно инсталлировать приложение из сети.

Диалоги по умолчанию

Оба режима инсталляции имеют три последовательности, в каждой из которых могут выводиться диалоги: **Start** (начало), **Progress** (развитие) и **End** (завершение). Ниже перечислены диалоги, использующиеся по умолчанию:

- Диалог **Welcome** (добро пожаловать) выводит пользователю приветственное сообщение. Имеется возможность заменить приветственный текст, выводящийся по умолчанию, на сообщение, задаваемое самим пользователем. В этот момент пользователь может либо прервать процедуру инсталляции, либо продолжить ее.
- С помощью следующего диалога — **Installation Folder** (папка инсталляции) — пользователь получает возможность выбрать папку, в которую должна производиться инсталляция. Если требуется добавить какие-либо нестандартные диалоги, то они должны быть включены перед этим диалогом.
- Диалог **Confirm Installation** (подтверждение инсталляции) — это последний диалог перед началом собственно инсталляции.

- Диалог **Progress** выводит управляющий элемент, который позволяет пользователю наблюдать за развитием процесса инсталляции.
- После завершения инсталляции выводится диалог **Finished** (завершение).

Рассмотренные диалоги по умолчанию будут выводиться автоматически, даже если мы вообще не будем открывать редактор пользовательского интерфейса. Для того чтобы в этих диалогах могли выводиться полезные для приложения-редактора сообщения, нам потребуется сконфигурировать упомянутые диалоги соответствующим образом.

Практикум: конфигурирование диалогов по умолчанию

1. Выберите диалог **Welcome**. В окне свойств вы увидите три свойства данного диалога, которые могут быть изменены: `BannerBitmap`, `CopyrightWarning` и `WelcomeText`. Выберите свойство `BannerBitmap`, нажав (**Browse...**) в комбинированном окне и выберите файл `wroxsetuplogo.bmp`, который мы записали в папку `Application Folder\Setup`. Этот рисунок будет выводиться в верхней части данного диалога.

   Текст, присвоенный свойству `CopyrightWarning` по умолчанию, гласит:

   ПРЕДУПРЕЖДЕНИЕ: Данная компьютерная программа защищена законом об авторских правах и международными соглашениями. Несанкционированное копирование или распространение этой программы либо любой ее части может привести к суровому гражданскому или уголовному наказанию и будет преследоваться в судебном порядке по всей строгости закона.

   Этот текст также будет выводиться в приветственном диалоге. Если он вам не подходит, то внесите в него необходимые изменения.

   Еще один текст, выводящийся в этом диалоге, определяется свойством `WelcomeText`:

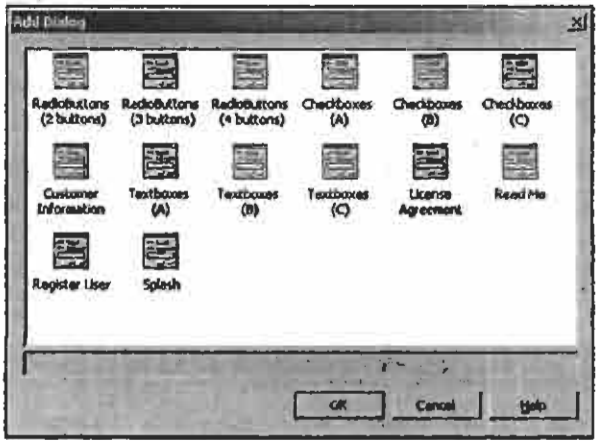
   Инсталлятор проведет вас через все этапы, необходимые для инсталляции на вашем компьютере продукта с именем `[НаименованиеПродукта]`.

   Этот текст также можно изменить. Строка `[НаименованиеПродукта]` автоматически заменяется на значение свойства `ProductName`, которое было опредено вместе с остальными свойствами проекта.

2. Войдите в диалог **Installation Folder**. Этот диалог обладает единственным свойством: `BannerBitmap`. Измените значение этого свойства точно таким же образом, как в диалоге **Welcome**. Этим свойством обладают все диалоги, при появлении которых выводится рисунок. Поэтому следует изменить значение этого свойства также и для всех остальных диалогов.

Дополнительные диалоги

Инсталлятор Visual Studio.NET не позволяет разрабатывать свои собственные диалоги и включать их в последовательность диалогов, выводящихся при инсталляции. Для выполнения таких действий вам потребуется использовать более сложные системы, например, InstallShield и Wise for Windows. Однако Visual Studio.NET



предоставляет возможность добавлять и настраивать большое количество предварительно определенных диалогов с помощью окна Add Dialog.

Выбрав в редакторе пользовательского интерфейса последовательность Start и перейдя в меню Action | Add Dialog, вы увидите окно Add Dialog, которое изображено на рисунке слева. Все входящие в это окно диалоги могут быть настроены.

На выбор предлагаются диалоги, в которых есть 2, 3 или 4 переключателя; диалоги с окнами выбора, в которые включено до 4 окон выбора; а также диалоги с текстовыми окнами, в которые входит до 4 текстовых окон. Управляющие элементы Windows для данных диалогов допускают настройку с помощью определения их свойств. В качестве примера того, каким образом можно использовать такие диалоги, воспользуемся диалогом Checkboxes.

Диалог Customer Information (информация о заказчике) запрашивает имя пользователя, компанию и серийный номер. Если у вашего продукта нет серийного номера, то вы можете отказаться от вывода серийного номера, присвоив свойству `ShowSerialNumber` значение `False`. Диалог License Agreement (соглашение о лицензии) предоставляет пользователю возможность принять условия лицензии до того, как начнется процедура инсталляции. Файл, в котором находится лицензия, определяется посредством свойства `LicenseFile`. В диалоге Register User (регистрация пользователя) пользователь может нажать кнопку Register Now (зарегистрироваться немедленно), после чего будет запущена программа, определяемая с помощью свойства `Executable`. Запускаемая программа может отправлять данные на FTP-сервер или пересылать данные по электронной почте. Диалог Splash (красочное пятно) просто выводит цветной экран перед началом инсталляции, при этом рисунок определяется посредством свойства `SplashBitmap`.

Практикум: добавление остальных диалогов

1. С помощью меню Action | Add Dialog добавьте диалоги Read Me, License Agreement и Checkboxes в последовательность Start. Определите очередность этих диалогов в стартовой последовательности, перенося их мышью в следующем порядке:

   Welcome – Read Me – License Agreement – Checkboxes – Installation Folder – Confirm Installation.

Все дополнительные диалоги должны располагаться перед диалогом Installation Folder, в противном случае при построении пакета инсталляции будет выдано соответствующее предупреждение.

2. Для всех этих диалогов необходимо настроить свойство `BannerBitmap` так же, как мы делали раньше. Свойству `ReadmeFile` диалога Read Me присвойте значение `readme.rtf` — файл, который мы предварительно поместили `Application Folder\Setup`.

3. Свойству `LicenseFile` диалога License Agreement присвойте значение `license.rtf`.

Свойства	Значения
BannerText	Optional Files (дополнительные файлы)
BodyText	Инсталляция дополнительных файлов
Checkbox1Label	Хотите ли вы иметь демонстрационный файл на рабочем столе?
Checkbox1Property	CHECKBOXDEMO
Checkbox2Visible	False
Checkbox3Visible	False
Checkbox4Visible	False

4. Диалог Checkbox будет использоваться, чтобы запросить у пользователя, требуется или нет инсталлировать файл `demo.wroxtext`, который был перенесен в `User's Desktop`. Измените свойства этого диалога в соответствии с таблицей слева.

Свойству `Checkbox1Property` присваивается то же самое значение, что и свойству `Condition` файла `demo.wroxtext`, которое было присвоено ранее при добавлении файла в пакет инсталляции с помощью системного редактора файлов. Если пользователь выберет это окно, то значение `CHECKBOXDEMO` будет равняться `true` и файл будет инсталлироваться; если окно выбрано не будет, то значение будет равняться `false` и файл соответственно инсталлироваться не будет.

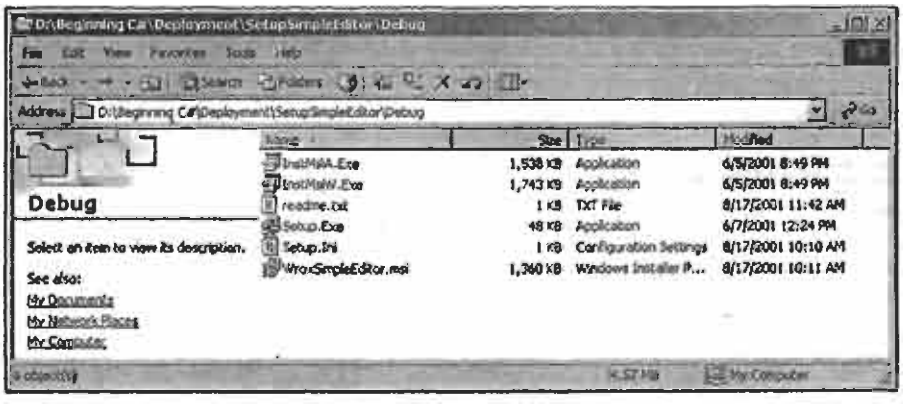
Свойству `CheckboxXVisible` остальных окон выбора присвоено значение `False`, поскольку нами используется только одно окно выбора.

Построение проекта

Теперь можно приступать к построению проекта по созданию инсталлятора.

Практикум: построение проекта

Для создания Microsoft Installer Package (пакета инсталляции компании Microsoft) выберите меню Build | Build Solution. Если построение проекта пройдет успешно, то можно обнаружить следующие файлы в директории `Debug` или `Release` (в зависимости от выбранных установок):

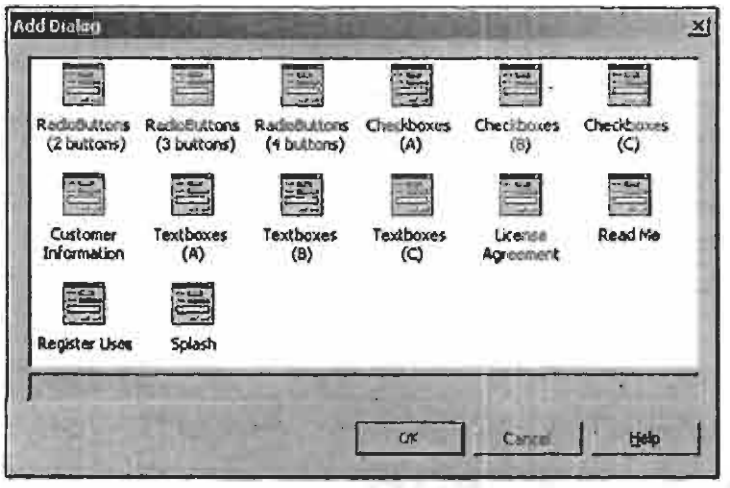


Файл `setup.exe` производит запуск инсталляции файла `WroxSimpleEditor.msi` БД MSI. Все файлы, помещенные в пакет инсталляции (за исключением одного), объединяются и подвергаются сжатию с получением MSI-файла, поскольку задано свойство Package Files in Setup File (упаковка файлов в файл начальной установки). Исключением из этого перечня является файл `readme.txt`. Для этого файла было изменено свойство `PackageAs` так,

чтобы он мог быть прочитан непосредственно перед началом процедуры инсталляции приложения. Собственно Windows Installer представлен файлами `InstMsiA.exe` *и* `InstMsiW.exe` *— ASCII-версия для Windows 98/ME и Unicode-версия для Windows NT4/2000/XP.*

Инсталляция

Теперь можно приступать к инсталляции приложения Simple Editor. Для запуска процедуры инсталляции следует либо два раза щелкнуть мышью на файле `setup.exe`, либо выбрать файл `WroxSimpleEditor.msi`, щелкнуть на нем правой кнопкой мыши и в открывшемся контекстном меню выбрать опцию **Install**, либо добиться того же результата в рамках Visual Studio.NET, в которой открыт проект инсталляции в Solution Explorer, как показано на рисунке слева.

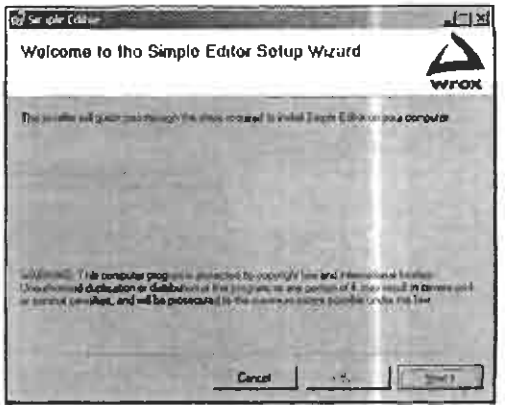


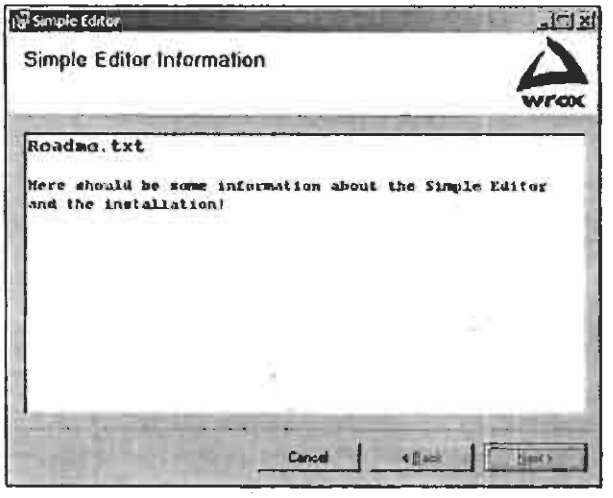
На всех приведенных ниже экранах присутствует логотип компании Wrox, а во вставленных дополнительно диалогах **Read Me** и **License Agreement** используются файлы, которые были указаны при их конфигурировании.

Просмотрим все три инсталляционные последовательности.

Диалог Welcome

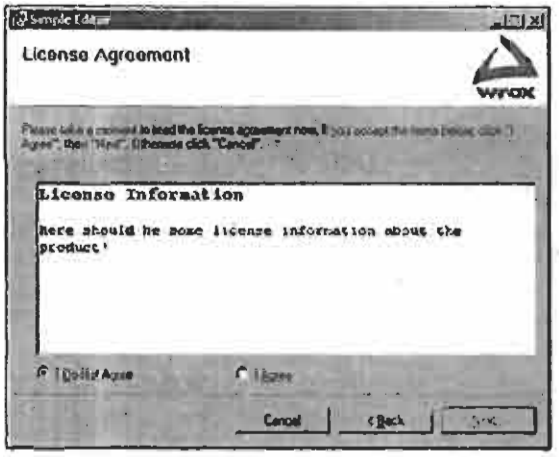
Первый диалог, который выводится на экран, — это диалог **Welcome**, помещен логотип издательства Wrox, который был включен нами с помощью присваивания значения свойству `BannerBitmap`. Название этого диалога получается в результате использования значения свойства `ProductName`, которое мы присвоили одновременно с остальными свойствами проекта (см. рис. справа).





Диалог Read Me

Нажав кнопку **Next**, мы увидим на экране сконфигурированный нами диалог **Read Me**. В названном диалоговом окне выводится файл `readme.rtf`, представленный в формате RTF, который был задан нами с помощью свойства `ReadMeFile`.



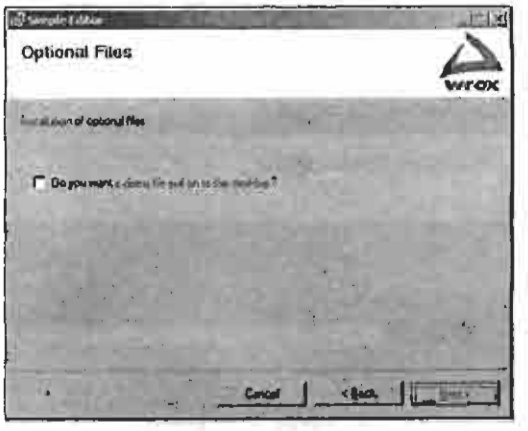
Диалог License Agreement

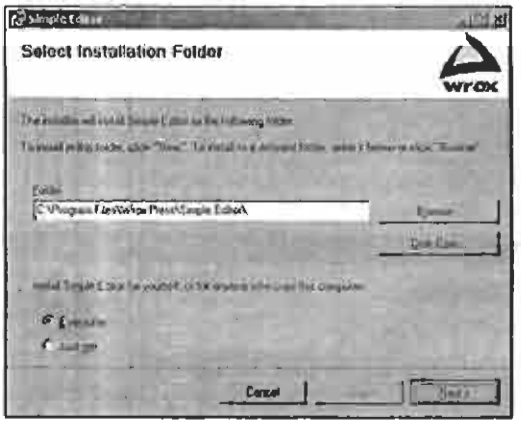
Третьим по счету будет выведен диалог соглашения о лицензии. В настоящем диалоге мы также сконфигурировали только свойства `BannerBitmap` и `LicenseFile`. Переключатели, которые используются для того, чтобы пользователь мог выразить свое согласие с условиями лицензии, добавляются автоматически. Как можно заметить на рисунке слева кнопка Next остается в нерабочем состоянии до тех пор, пока не будет выставлен флажок I agree (я согласен). Эта функциональная возможность включается в этот диалог автоматически.

Диалог Optional Files

Выражение согласия с информацией по лицензии и нажатие кнопки Next (см. рис. справа) приведет к выводу на экран диалога Checkbox. Для этого диалога мы не только настроили свойство, определяющее рисунок, но также присвоили текст свойствам `BannerText` и `BodyText` и `CheckBox1Label`. Все остальные окошки выбора остаются невидимыми, т. к. соответствующим свойствам `CheckboxNVisible` присвоено значение `False`.

Установка флажка в этом окошке выбора приведет к инсталляции файла `demo.wroxtext` на рабочем столе.



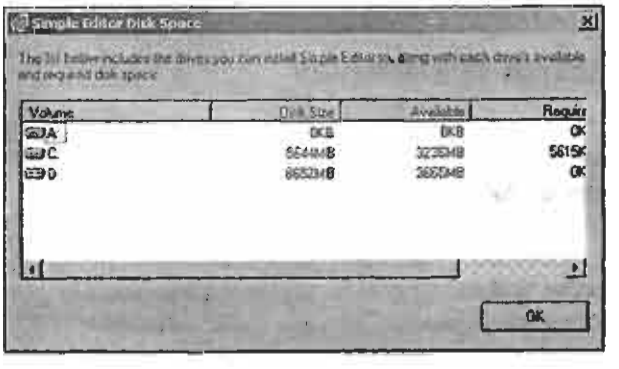


Диалог Select Installation Folder

С помощью этого диалога (см. рис. слева) можно задать путь, который будет использоваться при инсталляции приложения. В данном диалоге можно настраивать только свойство `BannerBitmap`. В качестве пути по умолчанию задан путь `[Program Files]\[manufacturer]\[Application Name]`.

Пользователь также имеет возможность определить, должно ли данное приложение инсталлироваться для всех пользователей или только для пользователя, зарегистрированного в системе в настоящий момент.

В зависимости от этого средство непосредственного вызова файла, содержащего программу, будет помещено либо в директорию данного пользователя, либо в директорию `All Users`.



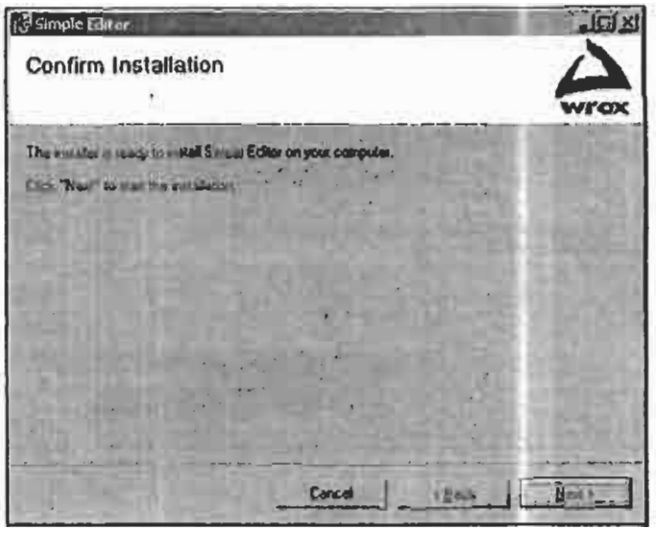
Кнопка Disk Cost

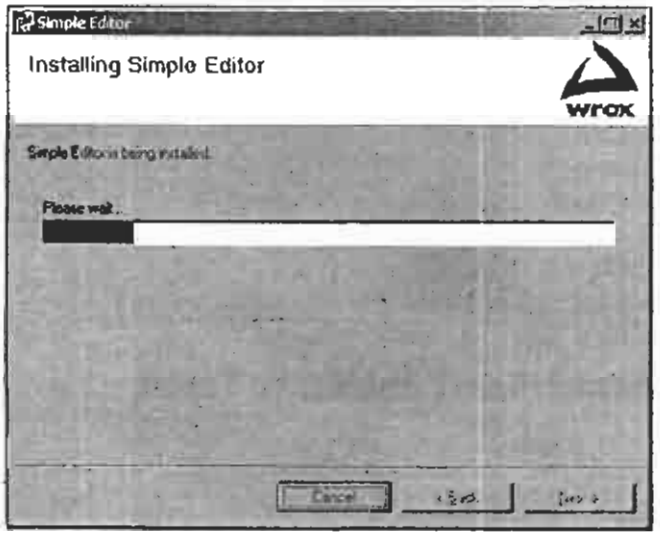
Нажатие кнопки Disk Cost (требуемая память на жестком диске) откроет диалог, изображенный на рисунке справа, в котором будет выведено суммарное дисковое пространство

и свободное пространство для каждого диска, а также необходимое пространство для осуществления инсталляции. Это помогает пользователю выбрать диск, на котором ему следует инсталлировать приложение.

Диалог Confirm Installation

Настоящий диалог является последним перед началом собственно процедуры инсталляции. В нем не задается больше никаких вопросов, это последняя возможность отменить процедуру инсталляции до ее начала (см. рис. справа).



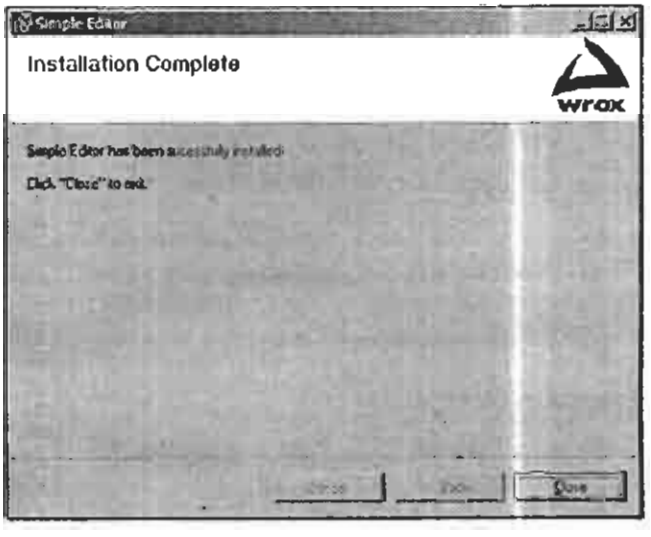


Диалог Progress

Диалог Progress выводит во время процедуры инсталляции управляющий элемент, который позволяет пользователю видеть, что выполнение процедуры идет, и предоставляет ему возможность приблизительно оценить, сколько времени она еще будет выполняться (см. рис. слева). Поскольку инсталлируемый редактор представляет собой небольшую программу, этот диалог будет закрыт очень быстро

Диалог Installation Complete

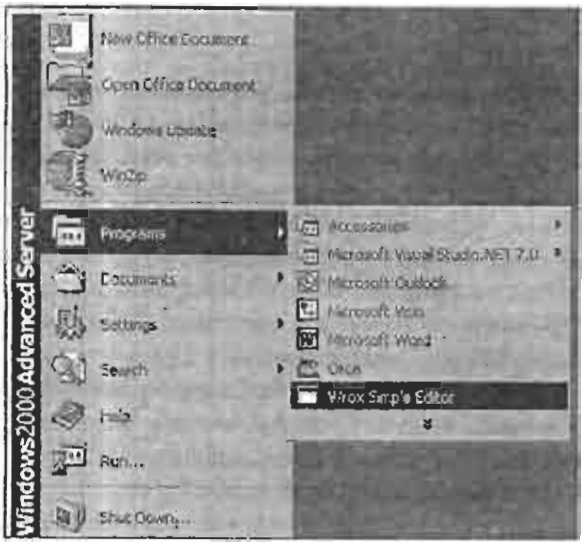
После успешного завершения процедуры инсталляции мы увидим последний диалог — Installation Complete (инсталляция завершена), представленный на рисунке справа.



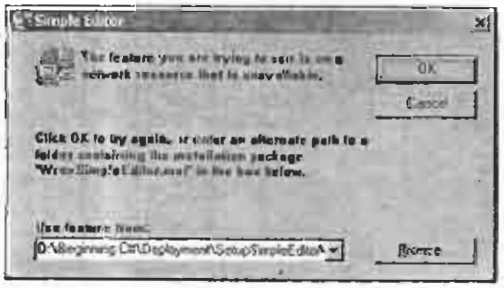
Запуск приложения

Приложение Simple Editor может быть запущено через вхождение меню Start | Programs | Wrox Simple Editor, как показано на следующем рисунке.

Поскольку мы зарегистрировали расширение файла, то есть и другая возможность запустить приложение: два раза щелкнуть



мышью на файле с расширением `.wroxtext` либо попытаться запустить редактор через пункт меню Start | Programs | Wrox Simple Editor. В этом случае восстановление будет запущено автоматически, т. к. будет обнаружено отсутствие обязательного файла! После начала процедуры восстановления вы увидите диалог, показанный на рисунке справа, если файлы начальной установки будут обнаружены там же, откуда осуществлялась инсталляция.



окно новый путь к папке, в которой содержится пакет инсталляции 'WroxSimpleEditor'.", "Следует использовать средство, расположенное:".

Обратная инсталляция

Если у вас возникнет желание избавиться от Wrox Simple Editor, вы сможете воспользоваться пунктом Add/Remove Programs, имеющемся в Control Panel, и нажать кнопку Remove для Simple Editor, чтобы вызвать соответствующее диалоговое окно, показанное на рисунке справа.

мышью на файле с расширением `.wroxtext`. Если выбрать соответствующее окошко в диалоговом окне Optional Files, то можно обнаружить файл `demo.wroxtext` на рабочем столе, в противном случае теперь можно создать этот файл самостоятельно с помощью только что инсталлированного инструмента — Wrox Simple Editor.

Самовосстановление

Simple Editor инсталлирован. Предположим, что файл почему-либо оказался испорчен. Чтобы выполнить следующее упражнение, автор уничтожил файл `C:\Program Files\Wrox Press\Simple Editor.exe`. Теперь можно войти в контекстное меню файла с расширением `msi` и выбрать пункт Repair (восстановление). Но можно воспользоваться более простым способом: щелкнуть два раза



В этом окне прочтем сообщение: "Идет подготовка к процедуре инсталляции...".

Если инсталляция осуществлялась с CD или директория инсталляции почему-либо более не доступна, то появится другой диалог (см. рис. слева), котором будет осуществлен запрос файла инсталляции. Упомянутое диалоговое окно содержит три сообщения: "Средство, которым вы пытаетесь воспользоваться, находится на недоступном сетевом ресурсе.", "Нажмите OK для повторения попытки либо введите в расположенное ниже

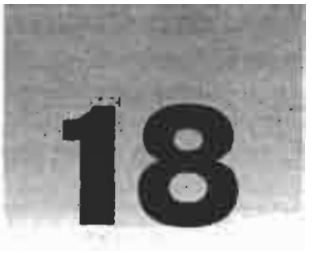


Итоги

В настоящей главе были рассмотрены функциональные возможности Windows Installer, а также то, каким образом можно создавать пакеты инсталляции с помощью Visual Studio.NET. Windows Installer существенно упрощает осуществление стандартизированных инсталляции, обратной инсталляции и восстановления.

Функциональные возможности Visual Studio.NET Installer несколько ограничены и не позволяют выполнять все действия, которые могут выполняться с помощью Windows Installer, однако для большинства приложений функциональных возможностей Visual Studio.NET Installer оказывается более чем достаточно. Существует большое количество редакторов, позволяющих сконфигурировать файл, генерируемый Windows Installer. Редактор системных файлов позволяет задавать все файлы и все способы непосредственного вызова, редактор условий запуска определяет некоторые предварительные условия, редактор типов файлов используется для регистрации расширений файлов для приложений, а редактор пользовательского интерфейса упрощает настройку диалогов, использующихся в процессе инсталляции.

При разработке приложений Windows наилучшим является создание пакета Windows Installer, и Visual Studio.NET позволяет сделать это очень просто.



Доступ к данным пользователя

Начиная с главы 3, в которой было введено понятие переменных и буквенных значений, мы уже осуществляли манипуляции с данными в C#. Однако до настоящего момента мы использовали только те данные, которые были "зашиты" в программах. Если такие данные изменялись, то нам приходилось вносить изменения непосредственно в саму программу. В настоящей главе мы собираемся рассмотреть то, каким образом можно осуществлять доступ к данным, хранящимся за пределами программ — в базах данных, распределенных документах и т. д.

Основное содержание любого бизнес-приложения — его данные. В любой компании хранятся данные, которые касаются ее служащих,— зарплата и выполняемая работа, информация о заказчиках и т. д. Большинству программ, которые вы будете писать в дальнейшем, будет требоваться осуществляемый тем или иным способом доступ к этим внешним данным независимо от того, является ли ваше приложение небольшой поисковой программой, позволяющей персоналу компании вводить и редактировать информацию о служащих в базе данных компании, или это полномасштабный web-сайт для ведения электронной коммерции (e-commerce), где активно используется каталог продукции и информация о заказчиках, например, номера их кредитных карточек и адреса доставки, которая хранится в базе данных на сервере.

В .NET Framework предусмотрен специальный набор объектов, находящихся в пространстве имен `System.Data`, которые обеспечивают возможность относительно простого доступа ко всем данным такого рода. В совокупности такие объекты известны под названием ADO.NET (это имя происходит от предшествующей технологии доступа к данным, которая называется ActiveX Data Objects — объекты данных ActiveX). ADO.NET будет подробно рассматриваться в следующей главе, а эта глава посвящена изучению того, как можно заставить Visual Studio.NET выполнять практически всю черновую работу за нас. Мы воспользуемся мощными возможностями Visual Studio (VS) для написания приложений, обрабатывающих данные. Это позволит познакомиться с наиболее важными понятиями, относящимися к работе с базами данных, использование которых позволяет обходиться без написания больших объемов кода. Если вы из тех, кто обожает писать код страницу за страницей, чтобы выполнить даже самое простое задание, то эта глава не для вас! А вот тем из вас, кто получает удовольствие, когда работа оказывается сделанной качественно и быстро, мы говорим: "Добро пожаловать!".

Доступ к данным в VS

ADO.NET является достаточно емкой темой, которой может быть посвящена отдельная книга, и прежде чем приступить к более детальному ее рассмотрению в следующей главе, нужно достаточно подробно изучить основные понятия, которые

позволят двигаться дальше. ADO.NET основывается на предшествующих ей технологиях доступа к данным и позволяет использовать их; поэтому начнем с рассмотрения ситуации, которая сложилась к моменту появления .NET.

Краткая история обеспечения доступа к данным

Когда были созданы самые первые БД, такие как Oracle и DB2 компании IBM, каждому разработчику, которому требовался доступ к хранящимся в них данным, приходилось использовать функции, специфические для конкретной базы данных. Каждая система обладала собственной библиотекой функций, например, для Oracle использовался Oracle Call Interface (OCI — интерфейс обращений к Oracle), а для SQL Server компании SyBase (позже купленной компанией Microsoft) — библиотека DBLib. Это обеспечивало быстрый доступ к данным, поскольку программы из этих библиотек работали с БД напрямую. С другой стороны, это означало, что программисты должны быть знакомы с библиотеками для каждой БД, с которыми им приходилось работать, поэтому задача написания приложений для работы с данными была весьма сложна. Это также означало, что, если компания переходила на использование другой БД, приложения приходилось переписывать заново.

Эта проблема была решена с помощью **Open Database Connectivity** (ODBC — **открытая система связи с базами данных**). Система ODBC была создана в начале 90-х гг. компанией Microsoft совместно с рядом других компаний и представляла собой стандартный набор функций, которые разработчики могли применять при работе с любыми базами данных. Эти функции транслировались в вызовы функций, присущих конкретной базе данных, с помощью драйверов, специфических для этой базы данных.

Это позволило решить основные проблемы, вызванные существованием различных библиотек баз данных,— разработчикам теперь нужно было уметь пользоваться только одним набором функций (функциями ODBC), и если компания переходила от одной БД к другой, то все, что им приходилось изменять,— это код, отвечающий за установление соединения с базой данных. При этом одна проблема все равно оставалась нерешенной. Хотя "безбумажный офис" по-прежнему оставался по большей части мифом, у компаний скопилось огромное количество представленных в электронном виде данных, хранящихся в самых разнообразных формах,— в виде электронных писем, web-страниц, файлов Project 2000 и т. п. ODBC замечательно работала с данными в традиционных базах данных, но не справлялась с другими типами данных, которые не хранились в виде аккуратных столбцов и строк и у которых могла отсутствовать какая-либо связная структура.

Решение этой проблемы привело к появлению **OLE DB**. OLE DB работает аналогично ODBC, представляя уровень абстракции между БД и приложениями, которым требуется доступ к данным. Клиентские приложения взаимодействуют с источником данных, который может представлять собой традиционную базу данных или любое другое хранилище информации, посредством провайдера OLE DB для этого источника. Данные, извлекаемые из источника, представляются в приложении в табличном формате — т. е. так, как будто они были извлечены из базы данных. OLE DB позволяет осуществлять доступ к данным, которые представляются существующими драйверами ODBC, и, таким образом, может быть использована для доступа к любым базам данных, поддерживающихся ODBC. ADO.NET поддерживает и OLE DB, и ODBC весьма сходным образом.

Последняя технология доступа к данным, о которой необходимо упомянуть,— это ActiveX Data Objects (ADO). ADO — это просто тонкий слой, который находится над OLE DB и позволяет программам, написанным на языках высокого уровня вроде Visual Basic, получать доступ к данным OLE DB.

Введение в ADO.NET

Хотя название ADO.NET и происходит от ADO, отличия между ними, на самом деле, весьма существенны. С точки зрения архитектуры эта система даже ближе к OLE DB. ADO.NET состоит из набора объектов, находящихся в пространстве имен `System.Data`, которые взаимодействуют с базами данных посредством **провайдеров данных .NET**. Объекты ADO.NET позволяют устанавливать соединения с базами данных, а также извлекать, редактировать, уничтожать и вставлять данные в базы данных и, кроме того, осуществлять манипуляции с данными программным путем.

Провайдеры данных .NET

ADO.NET состоит из двух фундаментальных частей — объекта `DataSet` и провайдера данных .NET. Объект `DataSet` используется для того, чтобы хранить в программе набор данных, представленных в табличном виде; при этом совершенно безразлично, откуда поступили эти данные. Провайдер данных состоит из набора компонентов, специфических для конкретного источника данных, которые позволяют устанавливать соединение и осуществлять взаимодействие с этим источником. Каждый провайдер находится в собственном пространстве имен в рамках пространства имен `System.Data`. На момент написания настоящей книги существовало три доступных провайдера данных:

- Провайдер данных для SQL Server. Он находится в пространстве имен `System.Data.SqlClient` и предназначается для работы с базами данных SQL Server версий 7.0 и выше, а также с базами данных MSDE (MSDE — это сокращенная версия SQL Server). Провайдер SQL Server является составной частью .NET Framework. Если возникает необходимость работы с базами данных SQL Server версий 6.5 или более ранних, то следует использовать провайдер OLE DB.
- Провайдер данных для OLE DB. Он используется для установления соединений с источниками данных посредством OLE DB и находится в пространстве имен `System.Data.OleDb`. Так же, как и провайдер SQL Server, провайдер OLE DB является составной частью .NET Framework. Провайдер OLE DB не может использоваться для установления соединений с базами данных посредством драйверов ODBC.
- Провайдер данных для ODBC. Он может использоваться для установления соединений с базами данных, обладающими драйверами ODBC. Провайдер данных ODBC находится в пространстве имен `System.Data.Odbc`. Он может быть загружен с web-сайта компании Microsoft, расположенного по адресу `http://download.microsoft.com/download/dasdk/Beta/1.0.2914.58/W98NT42KMeXP/EN-US/odbc_net.exe`.

Хотя на настоящий момент имеется возможность непосредственно устанавливать соединение только с БД SQL Server и MSDE, ожидается, что число доступных провайдеров данных будет увеличиваться с ростом числа пользователей .NET. Не так долго осталось ждать появления провайдера для Oracle. Но сейчас, при необходимости установить соединение с каким-либо другим источником данных вам придется воспользоваться либо OLE DB, либо ODBC.

Провайдеры данных состоят из четырех основных частей, описанных в следующей таблице:

Имя компонента	Описание
`Connection`	Используется для установления соединения с БД или каким-либо другим источником данных.
`Command`	Используется для извлечения, редактирования, уничтожения и включения данных в базу данных.
`DataReader`	Обеспечивает поток данных из источника данных. Такие данные могут только читаться (но не модифицироваться), и просмотр таких данных допускается только в прямом направлении.
`DataAdapter`	Используется для наполнения `DataSet` данными из источника данных, а также для обновления источника данных при внесении любых изменений в `DataSet`.

При использовании этих объектов не происходит обращения к ним по этим именам. Каждый провайдер обладает своей собственной реализацией этих классов, и, поскольку в классах каждого провайдера реализованы одинаковые интерфейсы, они все обладают одними и теми же методами и свойствами (хотя включение в класс какого-либо конкретного провайдера дополнительных методов и свойств, присущих только данному провайдеру, ничему не противоречит). Например, если требуется установить соединение с базой данных SQL Server, то для этих целей используется объект `SqlConnection`, а при необходимости соединения с источником данных ODBC — объект `OdbcConnection`. В обоих объектах реализуется интерфейс `IDbConnection`, поэтому они обладают одним и тем же набором методов и свойств.

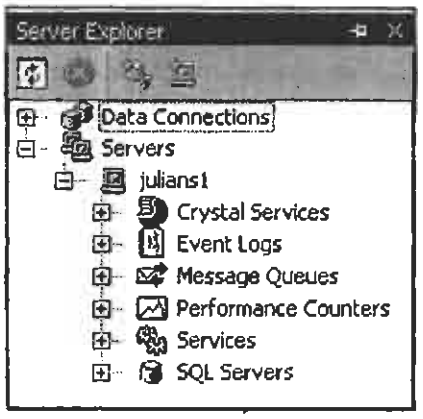
Объект DataSet

После установления соединения с БД и получения данных из нее предоставляется возможность выполнять различные манипуляции над данными с помощью объекта `DataSet`, представляющего собой семейство объектов `DataTable` (таблица данных). Объект `DataTable` позволяет хранить данные в табличном виде; т. е. в виде столбцов и строк. Объект `DataSet` не является составной частью провайдера данных (существует только один тип объекта `DataSet` независимо от того, поступили данные из источника данных SQL Server или OLE DB), следовательно, объект `DataSet` не отвечает за установление соединения с источником. Это означает, что когда мы выполняем какие-либо манипуляции над данными, находящимися в `DataSet`, то мы на самом деле работаем всего лишь с копией этих данных, расположенных в памяти локального компьютера. Это позволяет уменьшить нагрузку на сервер и на сеть, поскольку установление соединения с источником данных происходит только в момент первоначального обращения к данным и тогда, когда мы завершаем их редактирование и необходимо внести сделанные изменения в саму базу данных. Наряду с преимуществами, такая архитектура работы без установленного соединения приводит к возникновению определенных проблем. В среде с полностью разорванным соединением пользователь не может быть уведомлен об изменениях, которые были внесены в данные другими пользователями. Если для принятия решений используются данные, выведенные на экран, то всегда существует вероятность того, что эта информация уже устарела, если другие пользователи в этот момент вносят изменения в эти же самые данные.

Запускайте Visual Studio.NET, создавайте в папке `BegCSharp\Chapter18` новое C# Windows Application, присваивайте ему имя `Ch18Ex01` и приступайте к установлению соединения с базой данных!

Просмотр данных в VS

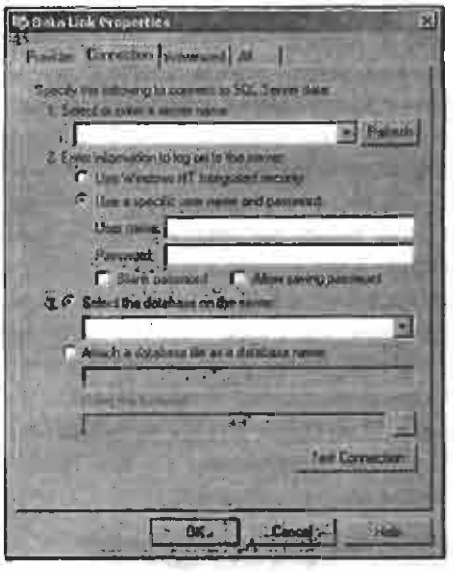
Прежде всего нам потребуется собственно база данных, с которой нужно установить соединение. В данном примере мы соединимся с базой данных SQL Server `Northwind`. Не беспокойтесь, если у вас нет SQL Server! Microsoft Data Engine (MSDE — средство для работы с данными компании Microsoft), которое поставляется вместе с Office и Access 2000, представляет собой не что иное, как сокращенную версию SQL Server, и с ним можно работать также, как если бы у вас была инсталлирована полная версия. MSDE не инсталлируется по умолчанию, и если MSDE отсутствует в вашей системе, то вы можете воспользоваться пошаговым руководством по инсталляции MSDE и импортированию базы данных `Northwind`, приведенном в приложении В.



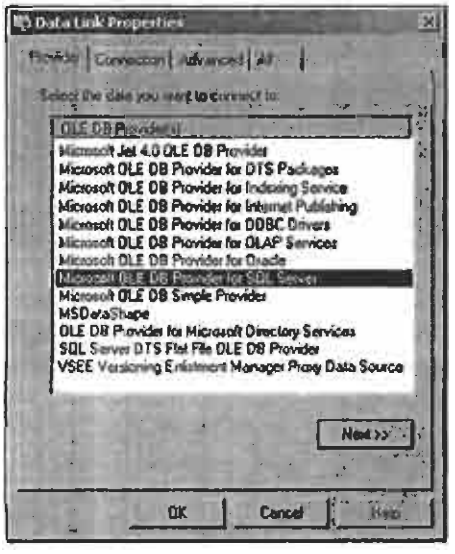
Установка соединения с базой данных

После открытия Visual Studio.NET, откройте **Server Explorer**, наведя курсор мыши на вкладку, расположенную с правой стороны экраиа (по умолчанию вкладка **Server Explorer** располагается непосредственно над вкладкой **Toolbox**). В названном окне (см. рис. слева) выводятся службы, которые имеются на локальной машине, а также службы, имеющиеся на любых доступных машинах. В данном случае нас интересует узел самого верхнего уровня в дереве просмотра **Server Explorer — Data Connections**, поскольку именно он позволяет включать в проект соединения с источниками данных.

Щелкните правой кнопкой мыши на узле **Data Connections** и выберите пункт **Add Connections...**. Откроется диалоговое окно **Data Link Properties** (свойства соединений с данными), представленное на рисунке справа.



По умолчанию провайдер соединения исходит из предположения о том, что мы будем осуществлять доступ к базе данных SQL Server, поэтому все выведенные на экран установки отиосятся к провайдеру OLE DB для SQL Server. Для



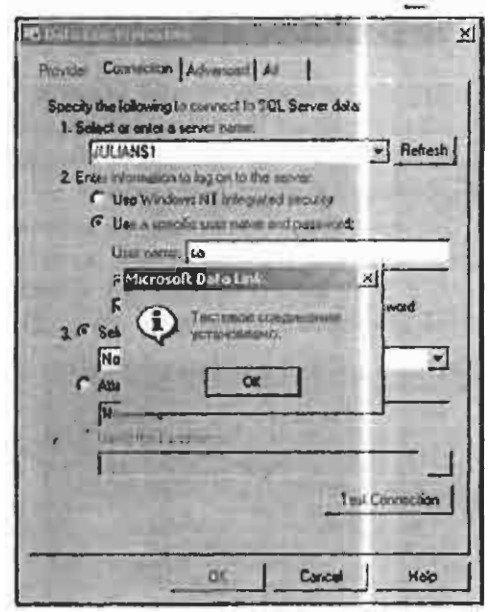
нашего примера потребуется образец БД `Northwind`, поэтому изменять установки нет необходимости. Однако если возникнет необходимость использовать какую-либо другую систему базы данных, то придется вернуться назад к вкладке **Provider** (см. рис. слева) и выбрать провайдер, который подходит для вашей базы данных.

Щелкнув мышью на вкладке Provider, вы получите перечень провайдеров, инсталлированных на системе. Поскольку нам необходим провайдер SQL Server, используемый по умолчанию, то следует просто нажать кнопку Next>>.

В настоящей главе будет использоваться SQL Server/MSDE и образец БД `Northwind`. Если же потребуется использовать какую-либо другую базу данных, то для этого просто выберите какой-либо подходящий для системы вашей базы данных провайдер OLE DB (например, для установления соединения с БД Access следует выбрать провайдер Microsoft Jet). Однако не следует путать эти провайдеры с провайдерами данных .NET. Data Link не позволяет выбирать провайдеры данных .NET, поэтому нам придется использовать провайдер данных OLE DB даже для SQL Server. VS, однако, оказывается в состоянии распознать, что соединение устанавливается с базой данных SQL Server, и создает объект `SqlConnection` самостоятельно.

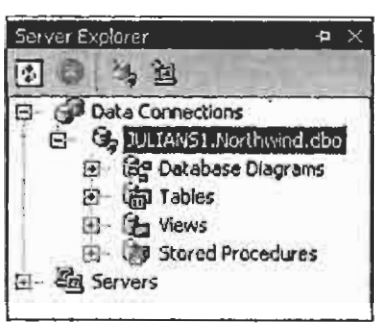
Щелкнув мышью на кнопке Next>>, мы вернемся обратно к вкладке Connection. Здесь потребуется задать всю информацию, необходимую VS для установления соединения с БД. Во-первых, необходимо либо ввести имя сервера, на котором инсталлирован SQL Server или MSDE, либо выбрать его из раскрывающегося списка.

Далее необходимо указать учетную запись пользователя, которую вы хотите использовать для установления соединения с базой данных. Если вы воспользуетесь пунктом Use Windows NT Integrated Security (использование объединенной системы безопасности Windows NT), то VS попытается установить соединение с БД с той же самой учетной записью, которую вы использовали для регистрации в Windows. Если для установления соединения с вашим SQL Server вы хотите воспользоваться другим именем пользователя и паролем, то необходимо выбрать следующую возможность — Use a specific username and password (использование заданного имени пользователя и пароля) и ввести необходимую информацию в соответствующие окна. По умолчанию при использовании как MSDE, так и SQL Server инсталлируется пользователь с именем `sa` и пустым паролем.



Наконец, выберите БД `Northwind` из раскрывающегося списка Select the database on the server (выбор БД на сервере). Если вы используете MSDE, а `Northwind` в списке отсутствует, то следует обратиться к приложению В за подробным описанием импорта требуемой базы данных.

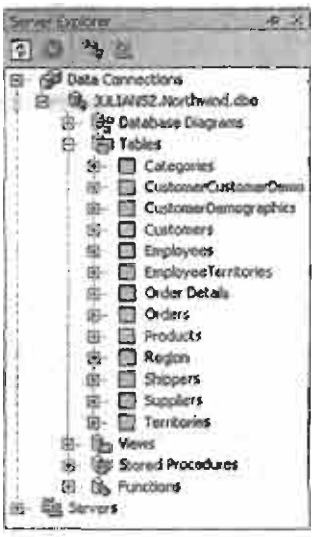
После ввода всей необходимой информации, щелкните мышью на кнопке Test Connection (проверочное соединение). Если будет выведено диалоговое окно с сообщением: "Тестовое соединение установлено." (см. рис. справа), то соединение установлено.



Примите наши поздравления — вы успешно установили свое первое соединение ADO.NET с базой данных! Теперь его можно увидеть — оно расположено непосредственно под Data Connections в Server Explorer (см. рис. слева).

Мы получили доступ к базе данных и ко всем хранящимся в ней данным даже не выходя из VS! Теперь стоит ознакомиться с их структурой. Это пригодится при написании кода, осуществляющего доступ к данным, чем мы будем заниматься в следующей главе.

Таблицы и отношения базы данных



Войдите в Server Explorer и откройте узел Tables (таблицы), который находится под только что добавленным соединением. Теперь можно увидеть список таблиц, имеющихся в базе данных (см. рис. справа).

Реляционные базы данных, к которым относятся SQL Server и Oracle, хранят данные в виде последовательности связанных между собой таблиц. Эти таблицы состоят из строк и столбцов; каждая строка представляет собой запись, хранящуюся в базе данных, а каждый столбец представляют собой отдельные поля каждой записи. Для того чтобы получить наглядное представление об этой структуре, щелкните правой кнопкой мыши на узле Customers (потребители) в Server Explorer и выберите пункт Retrieve Data from Table (извлечение данных из таблицы).

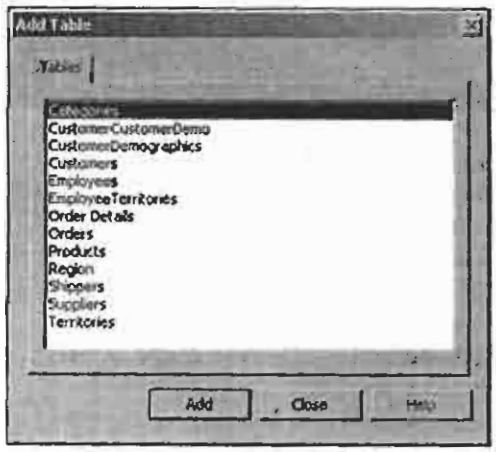
CustomerID	CompanyName	ContactName	ContactTitle	Address
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Co
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 231
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57
BLONP	Blondel père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléb
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	C/ Araquil, 67
BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bo
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen I
BSBEV	B's Beverages	Victoria Ashworth	Sales Representative	Fauntleroy Cir
CACTU	Cactus Comidas para llevar	Patricio Simpson	Sales Agent	Cerrito 333
CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager	Sierras de Gran
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Hauptstr. 29
COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate	Av. dos Lusíada
CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representative	Berkeley Garde
DRACD	Drachenblut Delikatessen	Sven Ottlieb	Order Administrator	Walserweg 21

Это приведет к тому, что VS загрузит и выведет на экран данные, хранящиеся в таблице `Customers` нашей базы данных `Northwind` (см. рис. слева).

Использующаяся в качестве образца база данных `Northwind` содержит данные о вымышленном торговце продуктами питания, снабжающем различные рестораны и продовольственные магазины. В таблице `Customers` хранится подробное описание каждого такого потребителя. Каждая строка этой таблицы соответствует отдельной компании, которую снабжает продуктами `Northwind`, а каждый столбец содержит конкретную информацию об этой компании: название компании, ее адрес, а также имя и должность лица, через которого осуществляется контакт с данной компанией.

Каждая строка таблицы определяется уникальным пятисимвольным идентификационным кодом, который хранится в поле `CustomersID`. Этот код известен под названием **основного ключа** и является жизненно важным с точки зрения установления отношений между таблицей `Customers` и другими таблицами в базе данных. Для того чтобы увидеть, как это работает, щелкните правой кнопкой мыши на узле Orders (заказы), и снова выберите пункт Retrieve Data from Table, на экран будет выведена необходимая нам таблица (см. рис. справа), в которой представлены заказы, полученные компанией Northwind. Каждая строка соответствует одному заказу. Заметьте, что в таблице также присутствует поле `CustomerID`, содержащее те же пятисимвольные идентификационные коды, что и в таблице `Customers`.

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate
10330	LILAS	3	16/11/1994	14/12/1994	28/11/1994
10331	BONAP	9	16/11/1994	28/12/1994	21/11/1994
10332	MEREP	3	17/11/1994	29/12/1994	21/11/1994
10333	WARTH	5	18/11/1994	16/12/1994	25/11/1994
10334	VICTE	8	21/11/1994	19/12/1994	28/11/1994
10335	HUNGO	7	22/11/1994	20/12/1994	24/11/1994
10336	PRINI	7	23/11/1994	21/12/1994	25/11/1994
10337	FRANK	4	24/11/1994	22/12/1994	29/11/1994
10338	OLDWO	4	25/11/1994	23/12/1994	29/11/1994
10339	MEREP	2	28/11/1994	26/12/1994	05/12/1994
10340	BONAP	1	29/11/1994	27/12/1994	09/12/1994
10341	SIMOB	7	29/11/1994	27/12/1994	06/12/1994
10342	FRANK	4	30/11/1994	14/12/1994	05/12/1994
10343	LEHMS	4	01/12/1994	29/12/1994	07/12/1994
10344	WHITC	4	02/12/1994	30/12/1994	06/12/1994
10345	QUICK	2	05/12/1994	02/01/1995	12/12/1994
10346	RATTC	3	06/12/1994	17/01/1995	09/12/1994

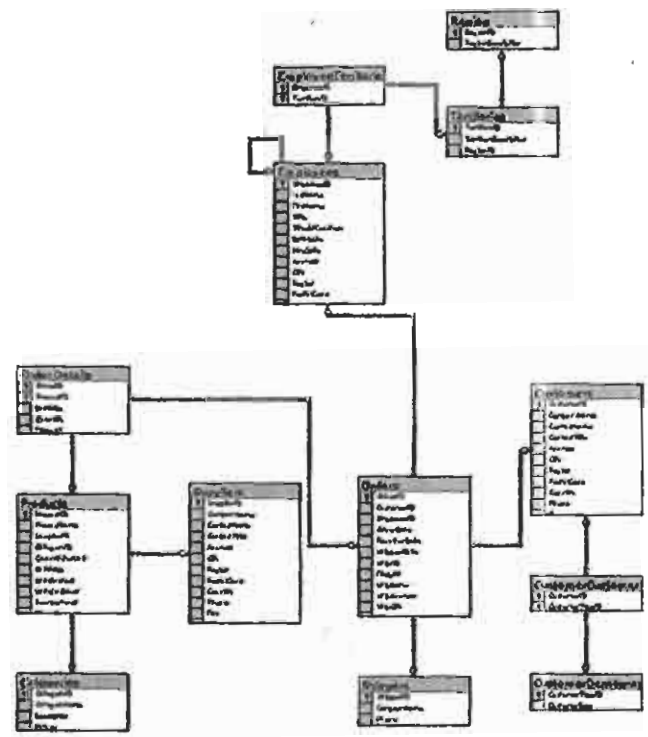


Значения, находящиеся в этом столбце, используются в качестве указателя на соответствующую строку в таблице `Customers`, в которой может быть найдена дополнительная информация о конкретном заказчике. Такой столбец известен под названием **внешнего ключа**.

В VS предусмотрен инструмент, позволяющий представлять эти взаимоотношения в рамках базы данных между основными и внешними ключами в виде диаграммы. Для того чтобы вывести диаграмму на экран, щелкните правой кнопкой мыши на узле **Database Diagrams** (диаграммы базы данных) соединения `Northwind` в **Server Explorer** и выберите пункт **New Diagram** (новая диаграмма). После этого VS выведет окно (см. рис. слева) — запрос относительно того, какие таблицы мы желаем включить в диаграмму.

Выберите все представленные в списке таблицы и щелкните мышью на кнопке **Add**. Затем щелкните мышью на кнопке **Close** и закройте диалог **Add Table** (добавление таблицы).

После этого VS создаст диаграмму, в которой будут изображены все заданные таблицы и отношения между ними (см. рис. справа).



Если вы используете MSDE и импортируете БД `Northwind` с использованием мастера Import and Export Data (импорт и экспорт данных), то на диаграмме не будут отражены отношения между таблицами, поскольку мастер не импортирует информацию об основных и внешних ключах.

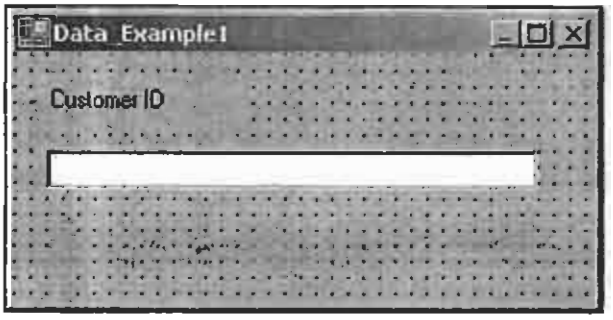
Для того чтобы получить на экране изображение всей диаграммы целиком, щелкните правой кнопкой мыши на каком-либо незанятом участке диаграммы и выберите пункт **Zoom | To Fit** из открывшегося меню. Хотя в этом случае, возможно, вы не сможете прочесть все названия столбцов! Отношения представлены в виде линий, соединяющих таблицы, причем на обоих концах линий имеются снмволы. Изображенне ключа, расположенное рядом с таблицей, свидетельствует о том, что основной ключ данной таблицы относится к внешнему ключу таблицы, находящейся на другом конце линии. Значок бесконечности обозначает внешнний ключ, который соответствует основному ключу соответствующей таблицы.

Вы, однако, покупали VS не для того, чтобы использовать ее в качестве клиентской части своей базы данных, поэтому познакомимся с тем, каким образом можно использовать установленное соединение из программ, написанных на C#!

Осуществление доступа к базе данных из приложения

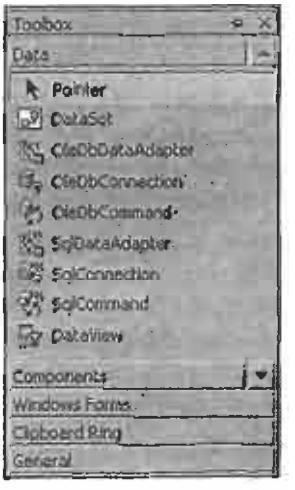
Вернитесь назад к окну разработки формы в проекте `Ch18Ex01`, добавьте в него текстовое окно и надпись и измените свойства этих объектов в соответствии с таблицей справа.

Управляющий элемент	Свойство	Значение
`Form`	`Name`	`frmMain`
	`Text`	`Data_Example1`
`Textbox`	`Name`	`txtCustID`
	`Text`	*(пусто)*
`Label`	`Name`	`lblCustID`
	`Text`	`Customer ID`

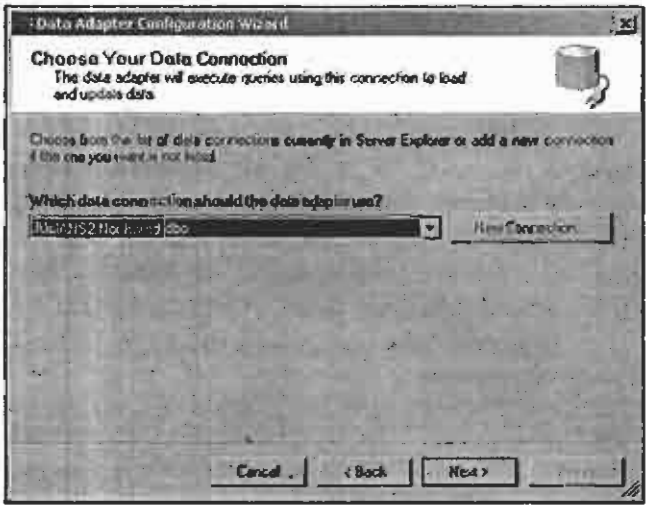


Окончательный результат должен выглядеть приблизительно так, как показано на рисунке слева.

Теперь добавим в проект соединение с базой данных. Откройте окно **Toolbox** (см. рис. справа) и выберите раздел **Data**. В этом меню выводятся все доступные управляющие элементы, позволяющие работать с данными.

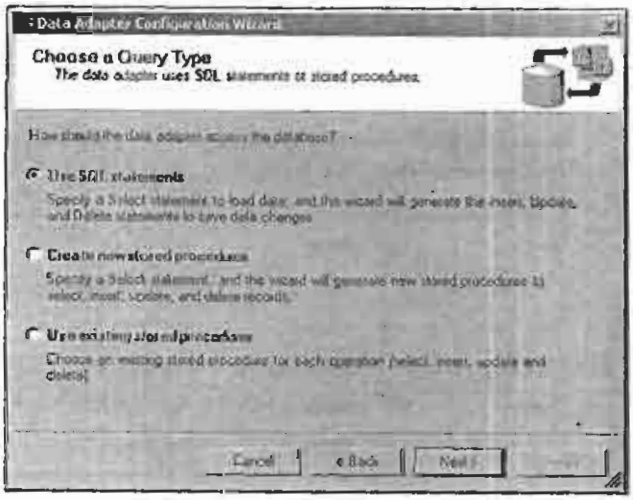


Существуют отдельные объекты `DataAdapter`, `Connection` и `Command` для провайдеров SQL Server и OLE DB. Для объектов SQL Server используется префикс `Sql`, а объекты OLE DB начинаются с `OleDb`. В данном случае, так как мы осуществляем доступ к БД SQL Server, будет использоваться провайдер SQL Server. Щелкните мышью два раза на `SqlDataAdapter` — откроется диалоговое окно мастера настройки конфигурации адаптера данных.



Мастер настройки конфигурации адаптера данных

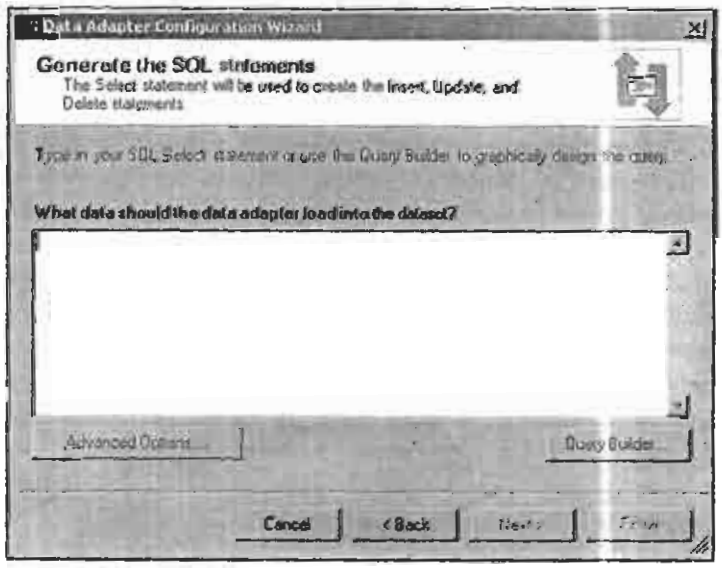
Появившись на экране, мастер настройки конфигурации адаптера данных задаст вопрос: какое из соединений с базами данных вы хотите использовать (см. рис. слева).



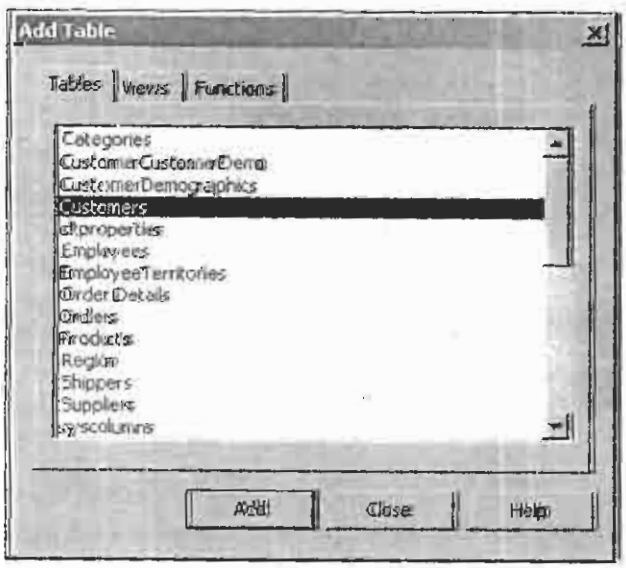
Есть единственное настроенное соединение с базой данных (настроенное ранее для доступа к базе данных `Northwind`), поэтому только оно и оказывается доступным в списке соединений. Если до настоящего момента ни одного соединения не установлено, т. е. возможность выбрать **New Connection...** (новое соединение) и создать его прямо здесь.

В следующем окне (см. рис. слева) мастер запрашивает, как мы будем определять данные, которые требуется извлечь.

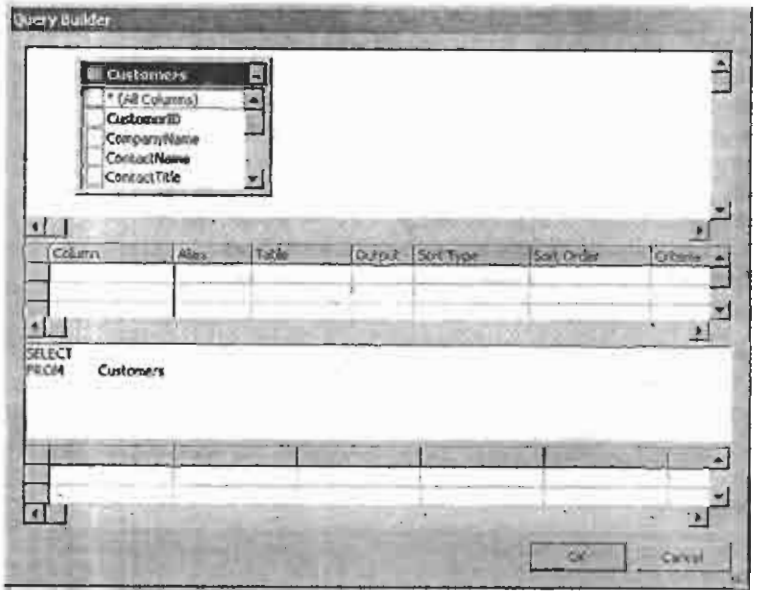
Выберите для нашего примера в диалоговом окне, представленном на рисунке слева, опцию **Use SQL Statements** (использование SQL-операторов). Эта опция делает возможным применение команд, написанных на Structured Query Language (SQL — структурированный язык запросов (не путать с SQL Server!)), для определения данных, которые требуется извлечь. SQL — это специальный язык, используемый для работы с реляционными базами данных.



Нажмите кнопку **Next>**, чтобы перейти к диалоговому окну для ввода SQL-оператора, с помощью которого будут извлечены необходимые данные (см. рис. справа). Однако совершенно не обязательно вводить SQL-оператор самостоятельно — это можно поручить VS.



Для того чтобы построить SQL-оператор с помощью VS, в том же диалоговом окне, в котором можно ввести оператор вручную, щелкните мышью на кнопке **Query Builder...** (построение запросов), расположенной в правом нижнем углу окна. Откроется диалоговое окно **Add Table** (см. рис. слева), с которым мы уже знакомы.



Теперь нам представлен список таблиц, хранящихся в базе данных Northwind. Выберите из него таблицу Customers. Щелкните мышью на кнопке **Add**. На экране появится диалоговое окно **Query Builder**, представленное на рисунке слева. Добавьте нашу таблицу Customers в **Query Builder**, а затем щелкните мышью на кнопке **Close**. Теперь у нас появилась возможность выбирать отдельные столбцы в таблице Customers.

Структурированный язык запросов

На данном этапе необходимо сделать несколько пояснений, касающихся SQL. Как мы уже отмечали ранее, SQL (допускаются различные варианты произношения — либо "сиквел", либо "эс-кью-эль" — на ваше усмотрение) — это язык, использующийся для работы с базами данных. SQL позволяет создавать комаиды, с помощью которых можно извлекать данные из баз данных, вставлять, уничтожать, а также редактировать данные. Более подробно язык SQL будет рассматриваться в следующей главе, однако в настоящий момент необходнмо познакомиться с командамн, которые мы будем использовать для извлечения данных из БД.

Операторы SELECT имеют следующий вид

```
SELECT столбец1, столбец2, ... столбецх FROM table (из таблицы),
```

например:

```
SELECT CompanyName, ContactName FROM Customers
```

Эта команда позволяет извлечь значения столбцов CompanyName и ContactName для всех строк таблицы Customers.

Существует также возможность ограничить число извлекаемых строк, добавив в оператор SELECT посылку WHERE. Предложение WHERE позволяет задавать значение конкретного столбца; в этом случае из базы данных будут извлекаться только те строки, у которых в соответствующем столбце имеется данное значение. Например,

```
SELECT CompanyName FROM Customers WHERE Region = 'WA'
```

Приведенный выше запрос позволяет извлечь названия компаний, расположенных вблизи Вашингтона.

Вместо указания имен столбцов можно использовать символ звездочки. Это позволит получить все столбцы, имеющиеся в таблице. Например:

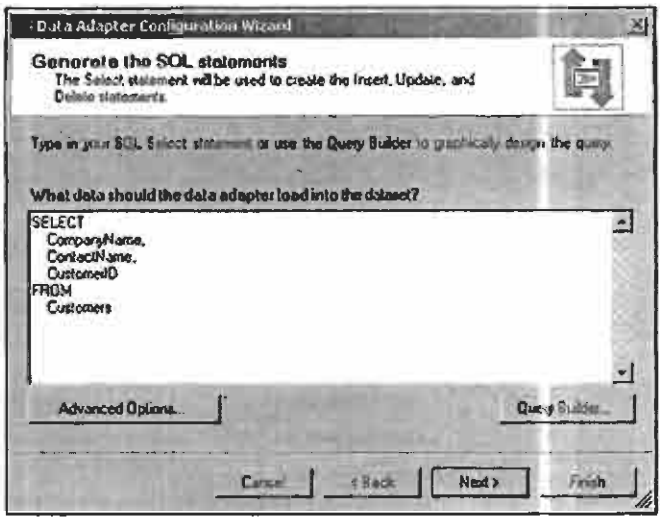
```
SELECT * FROM Customers
```

Этот оператор извлекает всю таблицу Customers целиком.

Будьте аккуратны при использовании звездочки. Хотя такая возможность является доступной и кажется весьма соблазнительной, почти все администраторы баз данных и программисты, работающие с базами данных, не советуют ею пользоваться. Использование символа * приводит к тому, что из базы данных извлекаются все столбцы, имеющиеся в данной таблице на момент выполнения запроса, а не только те, что были доступны на момент построения проекта. И если кто-нибудь добавит какие-либо поля в эту таблицу, то эти поля также будут извлекаться приложением наряду с полями, извлечение которых предполагалось изначально. Передача этой дополнительной информации может оказать существенное влияние на приложение.

Генерирование SQL-оператора

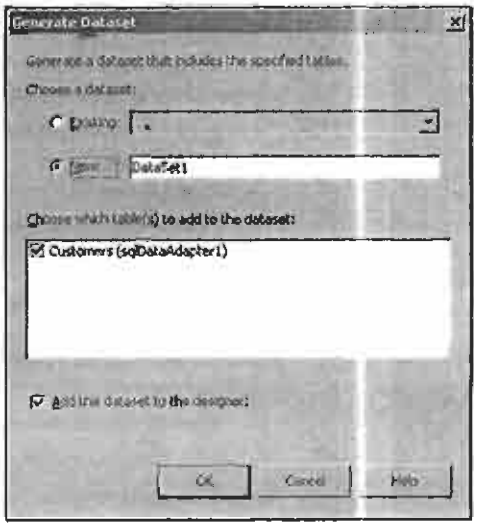
В этом примере потребуются столбцы `CompanyName`, `ContactName` и `CustomerID`. Выберите эти столбцы в окне `Customers` в Query Builder и нажмите OK. Query Builder в этом случае создаст SQL-оператор, который показан на рисунке справа.



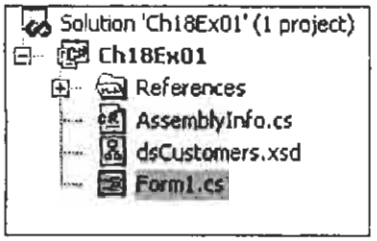
Нажмите кнопку Next>. Адаптер данных осуществит проверку всех установок и настроится на работу с приложением. По окончании нажмите кнопку Finish. Вы увидите, что в результате к форме были добавлены два новых объекта — `SqlDataAdapter` и `SqlConnection` (см. след. рис.).



В адаптере данных содержится основная информация о данных, которые мы хотим получить из БД. Адаптер данных можно представить себе в качестве мостика, соединяющего объект `DataSet` с базой данных. Адаптер данных, являющийся одним из компонентов провайдера данных .NET, используется для наполнения объекта `DataSet` данными, полученными из базы данных, а также для отсылки всех изменений, сделанных в `DataSet`, обратно в базу данных. В объекте `SqlConnection` содержится вся информация, необходимая ADO.NET для установления соединения с базой данных.



Теперь, когда мы располагаем адаптером и установленным соединением с базой данных, нам потребуется объект `DataSet`, что позволит работать с данными внутри приложения. Щелкните правой кнопкой мыши на только что созданном объекте `sqlDataAdapter1` и выберите опцию Generate Dataset... (создание объекта `DataSet`). В появившемся в результате этого диалоге (см. рис. справа) предоставлена возможность выбора: можно либо привязать адаптер данных к уже существующему объекту `DataSet`, либо создать новый.



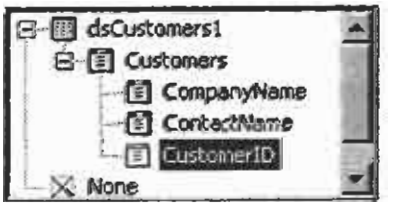
В нашем примере мы создадим новый объект `DataSet`. Поскольку у нашего проекта не имеется ни одного объекта `DataSet`, опция New (новый) будет выбрана по умолчанию. Измените имя создаваемого объекта `DataSet` на `dsCustomers` и нажмите OK.

Теперь появляется новый объект — `DataSet` — вслед за соединением и адаптером данных. Вы могли не заметить изменения в Server Explorer. Посмотрите внимательнее, в проекте добавлен новый файл с именем `dsCustomers.xsd` (см. рис. слева).

Названный файл представляет собой схему объекта `DataSet`. XSD-схема (XML Schema Definition — определение схемы XML) представляет собой документ, с помощью которого можно проверить правильность структуры XML-документа. XML — это текстовый формат, использующийся для представления данных, важность которого существенно возросла, в частности, в связи с распространением Интернета. XML и XSD-схемы будут рассматриваться в последнем разделе этой главы. Схема представляет весьма важный инструмент, использующийся ADO.NET, поскольку, организация и структурирование данных, выполняемые `DataSet` за кулисами, предполагают использование XML. Этот файл генерируется Visual Studio автоматически; в нем задается структура набора данных, все таблицы и все отношения между таблицами.

Поскольку объект `DataSet` уже создан, то мы получаем возможность привязать текстовое окно формы к данным, извлеченным из базы данных. Щелкните мышью на текстовом окне `txtCustID`, а затем прокрутите свойства этого окна до подраздела `DataBindings`. Если щелкнуть мышью на секции `Text` внутри `DataBindings`, то можно увидеть комбинированное окно, в котором находится `dsCustomers1` и все столбцы, имеющиеся в таблице. Выбрав в этом окне произвольный столбец, можно привязать соответствующее поле к этому текстовому окну. Это всего лишь означает, что по мере прокрутки набора данных значения, находящиеся в этом столбце, выводятся в этом окне. Все изменения, которые пользователь вносит в текстовое окно, отображаются также и в объекте `DataSet`. Объект `DataSet` не поддерживает постоянного соединения с источником данных; отсюда следует, что сам по себе этот источник не будет обновляться — до тех пор, пока мы не направим с этой целью соответствующий запрос.

В данном примере мы хотим привязать текстовое окно к столбцу `CustomerID`; для этого следует выбрать `CustomerID` в комбинированном окне (см. рис. справа).



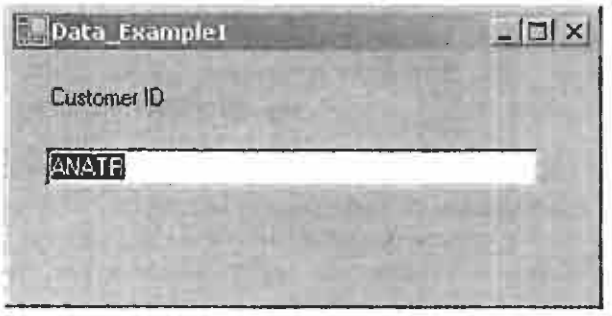
Если теперь вы сохраните и запустите свой проект, то увидите пустое текстовое окно. Почему это так? Разве те данные, с которыми мы работали на протяжении нескольких предшествующих страниц не должны оказаться в нашем распоряжении? Если отвечать на этот вопрос коротко, то нет, не должны. Не следует забывать, что до настоящего момента мы занимались лишь описанием той структуры, которую, как мы предполагаем, будут иметь данные, извлекаемые из таблицы. Мы пока ни разу не попытались описать реальные данные, кроме того, мы пока не давали команды приложению считывать какие-либо данные. Для этого потребуется включить в пример определенный код (на самом деле, это лишь малая часть из тех строк кода, который будет использоваться в данной главе).

Нам необходимо приказать адаптеру данных наполнить объект `DataSet`. Щелкните мышью дважды на пустом месте формы. После этого вы окажетесь в диалоге

редактирования кода, а курсор будет находится внутри обработчика события "загрузка формы". Введите в этот обработчик событий следующий код:

```
private void frmMain_Load(object sender, System.EventArgs e)
{
    // Наполнение набора данных данными, хранящимися
    // в базе данных Northwind в таблице "Customers"
    this.sqlDbDataAdapter1.Fill(this.dsCustomers1,1,0,"Customers");
}
```

В этих строках происходит обращение к методу `Fill()` адаптера данных. Этот метод осуществляет наполнение объекта `DataTable`, находящегося внутри `DataSet`, данными, полученными от источника данных. Этому методу передаются четыре параметра. Первый параметр — это объект `DataSet`, который мы собираемся наполнять, в данном примере — это набор данных `dsCustomer1`. Во-вторых, мы указываем, что его наполнение должно начинаться с первой записи. Третий параметр определяет количество требуемых записей. Если значение названного параметра равняется нулю, то это означает, что должны быть возвращены все имеющиеся записи. Наконец, мы указываем название таблицы, из которой необходимо извлекать данные.



Попробуйте еще раз сохранить и запустить данное приложение. Теперь вы должны увидеть значение поля `CustomerID` первой записи (см. рис. слева).

Вывод полной информации

То, что мы можем увидеть первое значение, — это просто замечательно, однако наверняка пользователям кроме этого потребуется узнать название фирмы и имя контактного лица, соответствующих данному ID, поэтому нужно добавить в форму несколько текстовых окон, перечисленных в таблице справа.

Управляющий элемент	Свойство	Значение
Label	Name	lblCustName
	Text	Customer Name (название заказчика)
Textbox	Name	txtCustName
	Text	(*пусто*)
	DataBindings/Text	dsCustomers1 – Customers.CompanyName
Label	Name	lblContactName
	Text	Contact Name
Textbox	Name	txtContactName
	Text	(*пусто*)
	DataBindings/Text	dsCustomers1 – Customers.CompanyName

Свойство `DataBindings` для обоих новых текстовых окон задается точно так же, как для самого первого текстового окна `txtCustID` — просто следует выбрать соответствующий столбец в комбинированном окне, которое расположено по соседству с подразделом `Text`, находящимся под свойством `DataBindings` данного текстового окна.

Ваша форма после этого должна принять вид, подобный изображенному на рисунке справа.



Если теперь запустить эту форму, то можно увидеть в ней дополнительную информацию, извлеченную из базы данных. На данном этапе можно добавлять (или оставлять) сколько угодно полей. Для нашего примера достаточно включить три столбца из таблицы, находящейся в адаптере данных. Если нужно выводить большее количество полей, то для этого следует щелкнуть правой кнопкой мыши на адаптере данных и выбрать опцию Configure Data Adapter... (настройка адаптера данных). В результате будет вызван тот же самый мастер Configure Data Adapter, которым мы пользовались ранее и который позволяет изменять число столбцов, извлекаемых из источника данных.

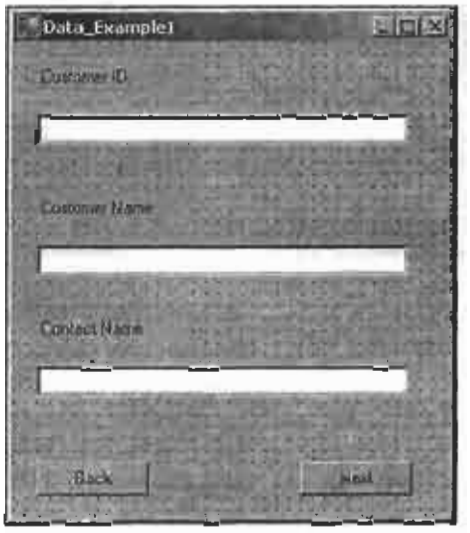
Перемещение по объекту DataSet

Мы добились вывода на экран первой записи, но пользователям может потребоваться увидеть следующую запись. И можно биться об заклад, что, увидев вторую запись, они захотят посмотреть третью. И нет никаких сомнений в том, что на этом они не остановятся!

Возможность перемещения по объекту `DataSet` оказывается чрезвычайно важной. .NET во всех смыслах основывается на использовании объектов, и наборы данных также представляют собой объекты, в которых содержатся таблицы, столбцы и строки в виде семейств. Перемещение по таким объектам очень напоминает перемещение по любому другому семейству — необходимо увеличивать на единицу счетчик, обозначающий текущую позицию в семействе.

Добавьте в форму объекты, входящие в таблицу слева.

Управляющий элемент	Свойство	Значение
Кнопка	Name	cmdBack
	Text	Back
Кнопка	Name	cmdNext
	Text	Next



В результате ваших действий упомянутая форма должна принять вид, подобный изображенному на рисунке справа.

Щелкните мышью два раза на кнопке `cmdNext` и добавьте следующий код в обработчик событий `Click`:

```
private void cmdNext_Click(object sender,
                           System.EventArgs e)
{
    // Переход к следующей записи в объекте DataSet из таблицы Customers
    this.BindingContext[this.dsCustomers1,"Customers"].Position++;
}
```

В этом коде для изменения текущей позиции используется объект BindingContext. Любая форма Windows обладает объектом BindingContext, который осуществляет контроль за всеми управляющими элементами данной формы. Увеличивая на единицу свойство Position объекта BindingContext для таблицы Customers нашего объекта DataSet, мы переходим к очередной записи в таблице, что приводит к выводу новых данных во всех текстовых окнах.

Аналогичным образом можно осуществлять движение и в обратном направлении. Введите следующий код для обработки события Click для кнопки cmdBack:

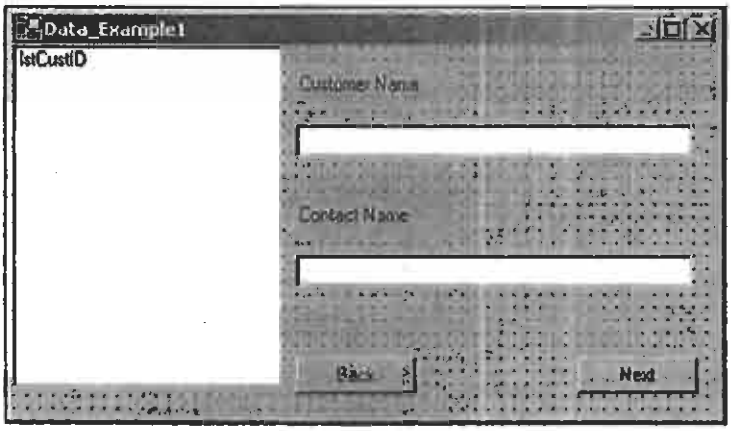
```
private void cmdBack_Click(object sender, System.EventArgs e)
{
    // Переход к предыдущей записи в объекте DataSet из таблицы Customers
    this.BindingContext[this.dsCustomers1,"Customers"].Position--;
}
```

Добавление списков

Добавить к форме окна со списками оказывается также просто, как и обычные текстовые окна. Добавьте к форме объект, описанный в таблице справа.

Управляющий элемент	Свойство	Значение
Окно со списком	Name	lstCustID
	DataSource	dsCustomers1.Customers
	DisplayMember	CustomerID

Узнав, как будет использоваться добавленное окно со списком для значений CustomerID, пойдем еще дальше



и удалим текстовое окно Customer ID. В результате форма должна получить вид, подобный представленному на рисунке слева.

После этого сохраните приложение и запустите его.

Теперь у нас есть окно со списком, в котором выводятся все значения CustomerID, имеющиеся в наборе данных. Выбор элементов списка не приводит к изменению положения указателя в BindingContext и, соответственно, к изменению данных, которые выводятся в текстовых окнах. Решение этой проблемы может показаться на первый взгляд весьма запутанным, однако оно является прекрасным примером объектно-ориентированного подхода к решению проблем.

Известно, что пользователю разрешается выбирать только один элемент списка. Семейства объекта обладают числовыми индексами. Каждый элемент списка представлен объектом, помещенным в окно со списком, и у каждого объекта имеется соответствующий численный номер. Если окно со списком будет заполняться записями в том же порядке, в каком они расположены в таблице DataSet, то численное значение индекса каждой записи будет совпадать с численными значениями индексов элементов списка. Кроме того, известно, что, для того чтобы переместить указатель в DataSet, необходимо указать новую позицию. Следовательно,

если бы можно было передавать индекс выбранного элемента списка в DataSet, то появилась бы возможность передвигать указатель и выводить в текстовых окнах необходимую запись.

Такой подход оказывается вполне работоспособным, и первым обработчиком события, к которому мы получаем доступ в результате двойного щелчка мышью на окне со списком в процессе разработки программы, является обработчик событий SelectIndexChanged. Это событие наступает в момент выбора элемента. Щелкните два раза мышью на текстовом окне и введите в обработчик событий следующий код:

```
private void lstCustID_SelectedIndexChanged(object sender, System.EventArgs e)
{
    // Передача объекту bindingContext нового значения индекса
    this.BindingContext[this.dsCustomers1,"Customers"].Position =
    this.lstCustID.SelectedIndex;
}
```

В этом случае также задается значение свойства Position объекта BindingContext, однако, вместо того чтобы увеличивать или уменьшать текущее значение индекса на единицу, мы просто передаем индекс элемента списка, который выбран в окне.

Требуется внести еще два изменения. Нажатие на управляющие кнопки не приводит к изменению элемента списка, выбранного в окне. И хотя указатель при этом соответствующим образом перемещается, эти перемещения не отображаются в окне со списком. Можно изменять индекс выбранного элемента списка точно так же, как ранее изменялась позиция в DataSet. Внесите следующие изменения в обработчик события "щелчок мышью" на кнопке cmdNext:

```
private void cmdNext_Click(object sender, System.EventArgs e)
{
    // Переход к следующей записи в объекте DataSet из таблицы Customers
    this.BindingContext[this.dsCustomers1,"Customers"].Position++;

    // Синхронизация указателей в управляющем
    // элементе ListBox и объекте BindingContext
    this.lstCustID.SelectedIndex =
        this.BindingContext[this.dsCustomers1,"Customers"].Position;
}
```

Добавленная строка программы, изменяет индекс выбранного элемента списка, присваивая ему новую позицию в DataSet. Другими словами, выбирается элемент списка, соответствующий текущей записи в DataSet. Аналогичные изменения могут быть внесены и для кнопки cmdBack:

```
private void cmdNext_Click(object sender, System.EventArgs e)
{
    // Переход к предыдущей записи в объекте DataSet из
    // таблицы Customers
    this.BindingContext[this.dsCustomers1,"Customers"].Position--;

    // Синхронизация указателей в управляющем
    // элементе ListBox и объекте DataSet
    this.lstCustID.SelectedIndex =
        this.BindingContext[this.dsCustomers1,"Customers"].Position;
}
```

Теперь, когда мы полностью определили пользовательский интерфейс, запустите проект и испытайте его.

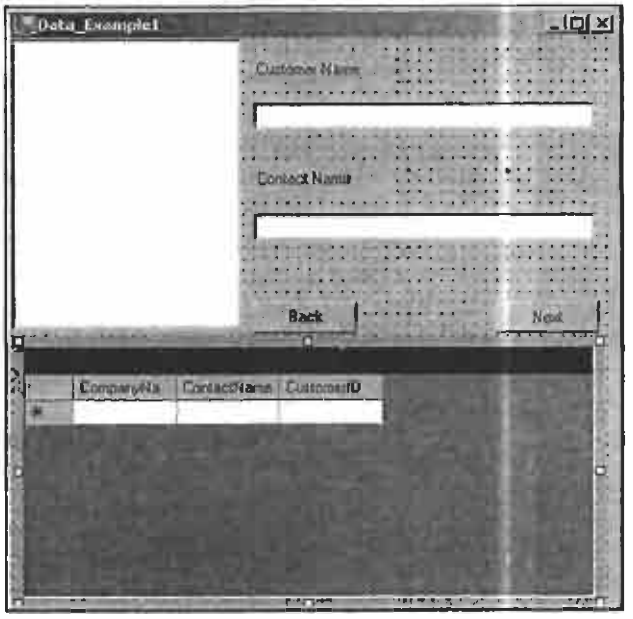
Добавление табличной сетки

Существует еще один элемент, который можно добавить. Хотя обычно табличные сетки не включаются в формы такой степени завершенности, как наша, они представляют собой очень быстрые и удобные объекты, которые могут использоваться в формах Windows для вывода данных и даже для внесения изменений в пользовательский интерфейс.

Управляющий элемент	Свойство	Значение
Табличная сетка	Name	dgCustomers
	DataSource	dsCustomers1.Customers

Добавьте к форме табличную сетку, свойства которой приведены в таблице слева.

Теперь форма должна принять вид, подобный показанному на рисунке справа.

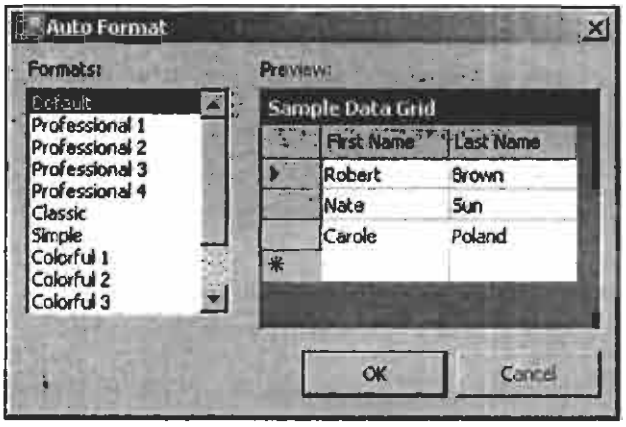


Единственное дополнительное изменение, которое требуется внести, для реализации привязки данных к табличной сетке,— это задать значение свойства `DataSource`; в данном конкретном случае мы присванваем ему значение `dsCustomers1.Customers`. Для данного примера больше никаких свойств изменять не требуется. Если вы используете табличную сетку без каких-либо других средств привязки, то вам потребуется сконфигурировать соединение и объект `DataSet` точно так же, как это было сделано настоящей главе ранее, и кроме того потребуется наполнить объект `DataSet`, обратившись к методу `Fill()`.

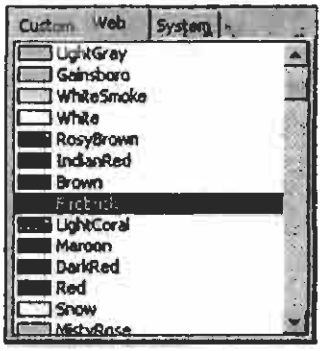
Сохраните приложение и запустите его. Все содержимое таблицы будет находится в табличной сетке.

Форматирование табличной сетки

Существует возможность изменять внешнее представление табличной сетки, позволяющее придавать приложениям более аккуратный вид. Наиболее простым способом изменения внешнего вида табличной сетки является использование многочисленных шаблонов, встроенных в объект. Например, если вас не устраивают стандартные цвета — голубой и серый, то вы можете внести изменения в шаблон; начните с того, что щелкните правой кнопкой мыши на табличной сетке и выберите Auto Format (автоформатирование).



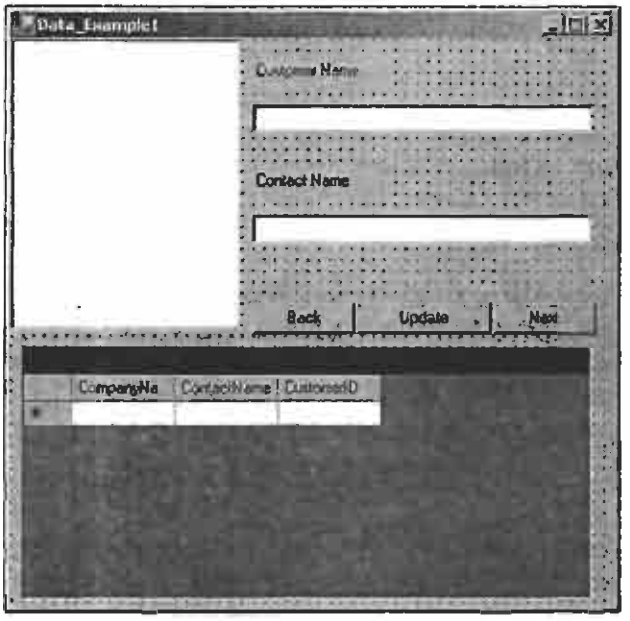
Откроется диалоговое окно Auto Format (см. рис. слева), в котором вы можете выбрать любой из предварительно определенных шаблонов для придания табличной сетке стройности. Однако если вы предпочитаете изменять ее внешний вид вручную, например,



если существует некоторая корпоративная схема использования цветов, которой нам необходимо следовать, то вы можете подкорректировать все свойства цветов фона и стилей шрифта.

Изменим фоновый цвет заголовка с голубого на красный. Прокрутите свойства табличной сетки до свойства `CaptionBackColor`. Если щелкнуть мышью на его текущем значении, то нам станут доступны три вкладки — Custom, Web и System (см. рис. справа).

В System содержатся заранее определенные цвета, используемые в Windows. Например, если выбрать в System цвет Desktop, то этот цвет будет изменяться, когда пользователь изменяет цвет своего рабочего стола. Иногда это может приводить к побочным эффектам, например, плохой читаемости. Поэтому при изменении этих значений следует проявлять максимум аккуратности. Мы в данном случае выбираем цвет Firebrick в вкладке Web.



Обновление базы данных

К настоящему моменту мы научились считывать данные из БД и вносить изменения в эти данные, находящиеся в текстовых окнах или табличной сетке, однако от этого будет не слишком много пользы, если у нас не будет возможности сохранить сделанные изменения в самой базе данных. Поэтому уделим немного внимания тому, каким образом данные в базе данных могут обновляться с помощью данных, которые хранятся в `DataSet`. На этот раз вместо использования мастера нам придется написать небольшой фрагмент программного кода. Добавьте к форме управляющую кнопку с именем `cmdUpdate`, чтобы форма приняла вид, подобный изображенному на рисунке слева.

Теперь добавьте следующий код в событие `cmdUpdate_Click`:

```
private void cmdUpdate_Click(object sender, System.EventArgs e)
{
    // Передача объекта DataSet обратно в базу данных
    this.sqlDbDataAdapter1.Update(this.dsCustomers1.Customers);
}
```

При нажатии этой кнопки в базе данных будут сохраняться любые изменения, внесенные в `DataSet`. Внесение изменений в базу данных более подробно рассматривается в следующей главе.

На момент написания книги функция `Update()` надежно работала со средствами привязки данных, только если для SQL Server использовался провайдер данных .NET; вызов функции `Update()` с провайдером OLE DB приводил к возникновению ошибок.

XML

До сих пор основное внимание уделялось реляционным базам данных, однако прежде чем мы завершим настоящую главу, мы должны сказать несколько слов о еще одном формате данных — **Extensible Markup Language** (XML, **расширяемый язык разметки**). XML — это способ хранения данных, представленных в простом текстовом формате, что означает, что эти данные могут быть прочитаны практически любым компьютером. Как мы увидим в одной из следующих глав, посвященных web-программированию, данное обстоятельство делает этот формат весьма подходящим для использования при передаче данных через Интернет. Этот формат допускает даже непосредственное прочтение человеком!

Состав этого языка может оказаться весьма сложным, поэтому мы здесь не будем рассматривать все его особенности. Сам же формат оказывается очень простым, и, по большей части, вам не потребуется детальное знание XML, поскольку обычно VS берет на себя выполнение основной работы — создавать XML-документы вручную вам придется в очень редких случаях. Отсюда следует, что XML чрезвычайно важен для .NET, поскольку он используется по умолчанию в качестве формата передачи данных; поэтому знание его основ представляется жизненно важным.

> *Если требуется получить более полное представление о XML, то вы можете обратиться к книге "Введение в XML" (Beginning XML), "ЛОРИ", 2003.*

XML-документы

Законченный набор данных известен в XML под названием **XML-документа**. XML-документ может представлять собой физический файл на вашем компьютере, а может быть всего лишь строкой в памяти. Он однако должен быть законченным и подчиняться определенным правилам. XML-документ состоит из нескольких различных частей, наиболее важными из которых являются **XML-элементы**, где содержатся собственно те данные, из которых состоит документ.

XML-элементы

XML-элементы состоят из открывающего тега (имени элемента, заключенного в угловые скобки, например, `<myElement>`), собственно данных самого элемента и закрывающего тега (который совпадает с открывающим тегом за исключением того, что у него имеется прямой слэш непосредственно после открывающей угловой скобки: `</myElement>`).

Например, мы могли бы определить элемент, в котором будет содержаться название некоторой книги, следующим образом:

```
<book>Tristram Shandy</book>
```

Если вы уже хотя бы немного знакомы с HTML, то вам может показаться, что это очень на него похоже — и вы окажетесь правы! На самом деле, и в HTML, и в XML используется во многом совпадающий синтаксис. Основное различие между ними заключается в том, что в XML не существует каких-либо заранее определенных элементов — можно самостоятельно выбирать имена для элементов, поэтому на количество существующих элементов не налагается никаких ограничений. Наиболее важным моментом, касающимся XML, о котором следует помнить,

является то, что несмотря на свое название XML, на самом деле, вовсе не является языком. Он представляет собой стандарт, который позволяет определять другие языки (известные как **XML-приложения**). У каждого такого языка имеется свой собственный словарь — специфическое множество элементов, которые могут быть использованы в документах, и некоторая структура, которую эти элементы могут принимать. Существует возможность явного ограничения элементов, наличие которых допустимо в XML-документе. Есть также возможность разрешить использование произвольных элементов, возложив задачу определения структуры документа на программу, которая его использует.

Имена элементов являются чувствительными к регистру, то есть `<book>` и `<Book>` представляют собой различные элементы. Это означает, что если вы попытаетесь закрыть элемент `<book>` с помощью закрывающего тега, в котором использование регистров не является идентичным открывающему тегу (например, `</Book>`), то такой документ не будет законным. Программы, которые осуществляют считывание XML-документов и анализ составляющих их элементов, известны под названием **XML-анализаторов**, и они отвергнут любой документ, в котором содержится незаконный XML.

Элементы могут содержать внутри себя другие элементы, поэтому можно модифицировать элемент `<book>` таким образом, чтобы он включал в себя автора и название книги в виде двух вложенных элементов:

```
<book>
   <title>Tristram Shandy</title>
   <author>Lawrence Sterne</author>
</book>
```

Однако перекрывание элементов не допускается, поэтому мы обязаны закрыть все субэлементы, до того как встретится закрывающий тег родительского элемента. Это, в частности, означает, что так поступать нельзя:

```
<book>
   <title>Tristram Shandy
   <author>Lawrence Sterne
   </title></author>
</book>
```

Это является недопустимым, поскольку элемент `<author>` открывается внутри элемента `<title>`, а закрывающий тег `</title>` располагается впереди закрывающего тега `</author>`.

Из правила, гласящего, что у всех элементов должен быть закрывающий тег, имеется одно исключение. Существует возможность использовать пустые элементы, внутри которых нет никаких данных или текста. В таком случае можно просто расположить закрывающий тег непосредственно вслед за открывающим тегом, как в примере, приведенном выше, либо использовать укороченную запись, включив слэш закрывающего элемента непосредственно в конец открывающего элемента:

```
<book />
```

Такая запись идентична полному синтаксису:

```
<book></book>
```

Атрибуты

Наряду с хранением данных внутри тела элемента существует возможность хранения данных с помощью атрибутов, которые помещаются внутри открывающего тега элементов. Атрибуты имеют следующий вид:

```
имя="значение",
```

причем значение *обязательно* должно быть заключено в одинарные или двойные кавычки. Например:

```
<book title="Tristram Shandy"></book>
```

или

```
<book title='Tristram Shandy'></book>
```

Обе эти конструкции вполне законны — в отличие от следующей:

```
<book title=Tristram Shandy></book>
```

На данном этапе вы, возможно, недоумеваете, зачем в XML понадобилось включать оба этих способа хранения данных. В чем заключается отличие между

```
<book>
   <title>Tristram Shandy</title>
</book>
```

и

```
<book title="Tristram Shandy"></book>
```

Если отвечать на этот вопрос честно, то следует признать, что между этими двумя способами не существует сколько-нибудь ощутимой фундаментальной разницы. Не существует также никаких преимуществ использования одного способа вместо другого. Элементы оказываются более подходящими, если существует вероятность того, что в дальнейшем потребуется включать дополнительную информацию, касающуюся этих данных, — в элемент всегда можно добавить субэлемент или атрибут — в отличие от атрибута, для которого это недопустимо. Спорным является утверждение о том, что элементы являются более удобочитаемыми и элегантными (в действительности, это дело личного вкуса каждого). С другой стороны, на атрибуты затрачивается меньшая полоса пропускания, если документ пересылается по сети без сжатия (при использовании сжатия большой разницы не существует); атрибуты также оказываются более удобными для хранения информации, которая не является существенной для каждого потребителя данного документа. Вероятнее всего, лучше использовать оба способа, каждый раз выбирая тот из них, который кажется вам наиболее удобным для хранения данного конкретного элемента информации. Простых и ясных правил на этот счет не существует.

XML-декларация

Помимо элементов и атрибутов XML-документы могут содержать в себе некоторое количество других составных частей (отдельные части XML-документа известны под названием **узлов**; т. е. элементы, текст внутри элементов и атрибуты — все это узлы XML-документа). Многие из них окажутся важными, только если вы действительно захотите изучать XML более углубленно. Однако существует один тип узла, который входит в состав практически в любого XML-документа. Этот узел — **XML-декларация**, и если мы включаем ее в документ, то она *должна* являться первым узлом этого документа.

XML-декларация по своему формату напоминает обычный элемент, однако внутрь угловых скобок XML-декларации включаются два вопросительных знака. Она всегда имеет имя `xml`, и в ней всегда присутствует атрибут `version`; на настоящий момент единственное возможное значение этого атрибута — `"1.0"`. Таким образом, простейшим возможным примером XML-декларации является следующий:

```
<?xml version="1.0"?>
```

Дополнительно в ней могут содержаться необязательные атрибуты `encoding` (кодировка — значение указывает на набор символов, которым следует пользоваться при чтении документа; например, `"UTF-16"` указывает, что документ использует набор 16-битовых символов в кодировке Unicode), и `standalone` (автономность — с возможными значениями `"yes"` и `"no"`, которые указывают соответственно на то, зависит или нет данный XML-документ от каких-либо других файлов). Эти атрибуты чаще всего не требуются, и вам обычно придется использовать в ваших XML-файлах только атрибут `version`.

Структура XML-документа

Одной из наиболее важных особенностей XML является тот факт, что он предоставляет способ структурирования данных, который очень сильно отличается от реляционных баз данных. Большинство современных систем баз данных предполагают хранение данных в таблицах, которые связаны между собой посредством значений, хранящихся в отдельных столбцах. В таблице данные хранятся в виде строк и столбцов — каждая строка представляет собой отдельную запись, а каждый столбец — отдельный блок информации, относящийся к данной записи. Напротив, XML-данные обладают иерархической структурой, отдаленно напоминающей структуру папок и файлов в Windows Explorer. У каждого документа *обязательно* должен иметься единственный **корневой элемент**, внутри которого содержатся все остальные элементы и текст. Если на самом верхнем уровне документа находится более одного элемента, то такой документ не является законным XML-документом, хотя *существует возможность* включать на самый верхний уровень другие XML-узлы, а именно XML-декларацию. Приведенный ниже документ является абсолютно законным:

```
<?xml version="1.0"?>
<books>
   <book>Tristram Shandy</book>
   <book>Moby Dick</book>
   <book>Ulysses</book>
</books>
```

В отличие от следующего документа:

```
<?xml version="1.0"?>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
```

Внутри этого корневого элемента существуют очень широкие возможности для определения структуры данных. В отличие от реляционных данных, у которых в каждой строке имеется одно и то же количество столбцов, в XML не существует никаких ограничений на количество субэлементов, которыми может обладать отдельный элемент. И хотя зачастую XML-документы обладают структурой, очень напоминающей структуру реляционных данных, в которой каждой записи соответствует отдельный элемент, XML-документы не должны обладать никакой заранее определенной структурой. Вследствие этого XML оказывается намного более приспособленным для хранения данных переменной структуры. Например, XML может быть использован для разметки текстовых документов. Язык web-разметки — HTML — хотя и не является, строго говоря, XML-приложением, но очень тесно связан с XML, как мы узнаем из главы 23.

Пространства имен XML

Любой программист может описывать собственные классы в C#; аналогично, каждый из нас может описывать собственные XML-элементы; это приводит к возникновению той же самой проблемы — как определить, какие элементы принадлежат каким словарям. Подобно тому, как в C# пространства имен используются для организации типов, пространства имен XML используются для описания XML-словарей. Такой подход позволяет включать в один XML-документ элементы сразу из нескольких различных словарей, не рискуя при этом перепутать элементы, например, если в двух различных словарях определяется элемент `<customer>`.

Пространства имен XML могут быть весьма сложными, поэтому в настоящий момент мы не будем обсуждать их очень подробно, а ограничимся основным синтаксисом, который весьма прост. Мы привязываем конкретные элементы или атрибуты к тем или иным пространствам имен с помощью префикса, за которым следует двоеточие. Например, элемент `<wrox:book>` представляет элемент `<book>` из пространства имен `wrox`. Но как узнать, что именно представляет собой пространство имен `wrox`? Для того чтобы такой подход оказался работоспособным, мы должны обладать возможностью гарантировать, что каждое пространство имен является уникальным. Наиболее простой способ добиться этого — привязать префиксы к чему-либо такому, что заведомо является уникальным. Именно такой подход и реализован: необходимо в каком-либо месте XML-документа привязать каждый используемый префикс пространства имен к **Uniform Resource Identifier** (URI — **единый идентификатор ресурса**). Существует несколько разновидностей единых идентификаторов ресурса, но в качестве наиболее распространенного используется просто web-адрес, вроде `"http://www.wrox.com"`.

Для того чтобы некоторому префиксу поставить в соответствие конкретное пространство имен, следует поместить внутрь элемента атрибут `xmlns:<префикс>`, присвоив ему в качестве значения уникальный URI, определяющий данное пространство имен. После этого данный префикс может быть использован в любом месте элемента, включая все вложенные в него элементы. Например:

```
<?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>
```

В данном случае использование префикса wrox: в элементах `<title>` и `<author>` является допустимым, поскольку они расположены внутри элемента `<book>`, в котором описывается данный префикс. Однако если бы мы попытались включить этот префикс в элемент `<books>`, то это привело бы созданию незаконного XML-документа, поскольку для данного элемента этот префикс не описан.

Есть также возможность описывать пространство имен, используемое по умолчанию, с помощью атрибута `xmlns`:

```
<?xml version="1.0"?>
<books>
   <book xmlns="http://www.wrox.com">
      <title>Beginning C#</title>
      <author>Karli Watson</author>
      <html:img src="begsharp.gif"
                xmlns:html="http://www.w3.org/1999/xhtml"/>
   </book>
</books>
```

В этом примере пространство имен, используемое по умолчанию в элементе <book>, определено как "http://www.wrox.com". Все, что входит в состав данного элемента, будет принадлежать объявленному пространству имен за исключением тех случаев, когда будет явно указано на необходимость использовать другое пространство имен, как это сделано для элемента <img> (мы относим его к пространству имен, которое используется для XML-совместимых HTML-документов).

Правильно оформленный и допустимый XML

До сих пор мы использовали понятие "законного" XML. На самом деле в XML проводится различие между двумя формами "законности". Документы, которые подчиняются всем требованиям, предъявляемым стандартом XML, называются **правильно оформленными**. Если какой-либо XML-документ не является правильно оформленным, то анализаторы будут лишены возможности проинтерпретировать его корректно и отвергнут такой документ. Для того чтобы документ мог считаться правильно оформленным, он должен:

- Обладать одним и только одним корневым элементом
- Обладать закрывающими тегами для всех своих элементов (за исключением случаев использования сокращенного синтаксиса, упоминавшегося выше)
- Не иметь перекрывающихся элементов — все дочерние элементы должны быть полностью вложенными в родительские
- Использовать только атрибуты, заключенные в кавычки

Этот перечень ни в коем случае не является исчерпывающим, однако он содержит в себе наиболее распространенные ловушки, в которые попадают программисты, только начинающие знакомиться с XML.

Однако XML-документы могут подчиняться всем этим правилам и при этом все равно не быть **допустимыми**. Сам по себе XML не является языком программирования, это — стандарт для описания XML-приложений. Правильно оформленные XML-документы — это всего лишь документы, соответствующие стандарту XML; для того чтобы являться допустимыми, они, кроме того, должны подчиняться всем правилам, относящимся к данному XML-приложению. Не все анализаторы выполняют проверку допустимости; анализаторы, выполняющие такую проверку, называются **анализаторами с проверкой допустимости**.

Чтобы иметь возможность проверять документ на предмет его соответствия правилам, относящимся к данному приложению, прежде всего, необходимо обладать способом определения таких правил.

Проверка допустимости XML-документов

XML поддерживает два различных способа определения того, какие элементы и атрибуты могут быть включены в документ и в каком порядке — **Document Type Definitions** (DTD — **определение типа документа**) и **Schemas** (**схема**). В определениях типа документа используется синтаксис, отличный от XML синтаксиса, который был унаследован от прародителей XML и который теперь постепенно вытесняется схемами. DTD-определения не позволяют определять типы данных в элементах и атрибутах, поэтому они оказываются недостаточно гибкими и в контексте .NET Framework используются достаточно редко. Схемы оказываются более

гибкими и позволяют задавать типы данных, кроме того, они созданы с использованием синтаксиса, совместимого с XML. К сожалению, схемы очень сложны и существуют в нескольких различных форматах даже в рамках .NET!

Схемы

В .NET поддерживаются два различных формата схем — XML Schema Definition Language (XSD — язык описания XML-схем) и XML-Data Reduced schemas (XDR — упрощенные схемы XML-данных). Схемы могут быть включены либо в состав самого XML-документа, либо храниться в виде отдельного файла. Эти два формата являются несовместимыми, и, для того чтобы воспользоваться любым из них, надо, действительно, очень хорошо знать XML. По этой причине мы не будем разбирать эти форматы подробно. С другой стороны, полезно уметь распознавать основные элементы схем, поэтому основные принципы построения схем будут объяснены. С этой целью мы рассмотрим пример XSD- и XDR-схем для некоторого простого XML-документа, в котором содержится основная информация о паре книг, вышедших в издательстве Wrox:

```
<?xml version="1.0"?>
<books>
   <book>
      <title>Beginning C#</title>
      <author>Karli Watson</author>
      <code>4982</code>
   </book>
   <book>
      <title>Professional C#</title>
      <author>Simon Robinson</author>
      <code>4990</code>
   </book>
</books>
```

XSD-схемы

Элементы, входящие в состав XSD-схем, должны принадлежать пространству имен `"http://www.w3.org/2001/XMLSchema"`. Если это пространство имен не будет объявлено, то элементы схем не будут распознаваться.

Чтобы привязать к документу XSD-схему, находящуюся в отдельном файле, необходимо в корневой элемент документа включить элемент `schemaLocation`:

```
<?xml version="1.0"?>
<books schemaLocation="file://C:\Csharp\xml\books.xsd">
   . . .
</books>
```

Рассмотрим пример XSD-схемы:

```
<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, на что следует обратить внимание в этой схеме, это то, что в ней в качестве пространства имен, используемого по умолчанию, задается пространство имен XSD. Это позволяет анализатору исходить из того, что все элементы данного документа принадлежат схеме. Если это пространство имен не будет задано, то анализатор будет воспринимать эти элементы просто как обычные XML-элементы и не поймет, что их следует использовать для осуществления проверки допустимости.

Вся схема целиком содержится внутри элемента с именем `<schema>` (с буквой `s`, набранной в нижнем регистре — не забывайте, что регистр имеет значение!). Все элементы, которые могут находиться внутри документа, должны представляться элементами `<element>`. У этого элемента имеется атрибут `name`, в котором хранится имя элемента. Если элемент может содержать в себе вложенные дочерние элементы, то необходимо для каждого из них включить теги `<element>` в элемент `<complexType>`. Внутри этого элемента можно определить, каким образом должны располагаться дочерние элементы. Например, чтобы указать, что в элементе могут присутствовать дочерние элементы в произвольном порядке, следует использовать элемент `<choice>`, а для того чтобы указать, что дочерние элементы должны присутствовать в том же самом порядке, в котором они перечислены в схеме, необходимо использовать элемент `<sequence>`. Если некоторый элемент может появляться более одного раза (например, элемент `<book>`), то в родительский элемент следует включить атрибут `maxOccurs`. Если присвоить этому атрибуту значение `"unbounded"` ("не ограничен"), то это будет означать, что дочерний элемент может встречаться произвольное колнчество раз. И, наконец, каждый атрибут должен представлен элементом `<attribute>`, включая использованный нами ранее атрибут `schemaLocation`, с помощью которого анализатору передается информация о местонахождении схемы. Этот элемент располагается в конце списка дочерних элементов.

XDR-схемы

Чтобы привязать внешнюю XDR-схему к XML-документу, необходимо указать для документа пространство имен со значением `"x-schema:<schema_имяфайла>"`:

```
<?xml version="1.0"?>
<books xmlns="x-schema:books.xdr">
   . . .
</books>
```

Схема, приведенная ниже, представляет собой XDR-эквивалент XSD-схемы, рассматривавшейся выше. Эти схемы существенно отличаются друг от друга:

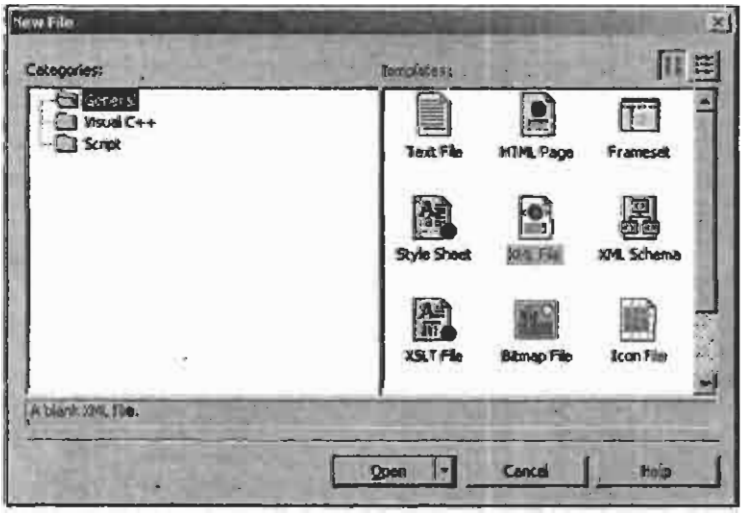
```
<Schema xmlns="urn:schemas-mucrosoft-com:xml-data">
   <ElementType name="title" content="textOnly" />
   <ElementType name="author" content="textOnly" />
   <ElementType name="code" content "textOnly" />
   <ElementType name="book" content "eltOnly">
      <group order="seq">
         <element type ="title" />
         <element type="author" />
         <element type=code" />
      </group>
```

```
    </ElementType>
    <ElementType name="books" content="eltOnly">
       <element type="book" />
    </ElementType>
</Schema>
```

В этом случае мы также задаем пространство имен, используемое по умолчанию, чтобы сообщить анализатору, что все элементы, входящие в состав документа, относятся к определению схемы; в данном случае — это пространство имен `"urn:schemas-microsoft-com:xml-data"`. Обратите внимание на то, что (в отличие от XSD-схем) на этот раз используется свой собственный формат, который вообще не будет работать ни с какими продуктами, кроме созданных компанией Microsoft. На самом деле, XDR-схемы оказываются особенно полезными при работе с SQL Server — сервером базы данных компании Microsoft, поскольку у них имеется встроенная поддержка XDR.

На этот раз в качестве корневого элемента мы используем элемент `<Schema>`, начинающийся *с заглавной буквы* 'S'. В данном случае корневой элемент также содержит в себе все описание схемы полностью (не забывайте о том, что XML-документы должны обладать единственным корневым документом). Однако далее все идет совсем по-другому — элементы, которые будут появляться в нашем документе, описаны *в обратном порядке*! Причина такого расположения элементов заключается в том, что каждый элемент документа представляется в схеме элементом `<ElementType>`, в котором, в свою очередь, содержатся элементы `<element>` (обратите внимание, что в данном случае используется строчная буква 'e') для каждого дочернего элемента. Внутри тегов `<element>` нами задается атрибут `type`, который должен указывать на `<ElementType>`, следовательно, он должен быть к этому моменту уже определен. Если нужно наложить ограничения на то, каким образом могут располагаться дочерние элементы, то можно воспользоваться элементом `<group>`, располагающимся внутри элемента `<ElementType>`, и задать его атрибут `order`. В данном случае, ему присваивается значение `"seq"`; это означает, что в документе элементы должны располагаться в том же порядке, в котором они располагаются в схеме,— абсолютно аналогично тегу `<sequence>` в XSD-схеме.

Практикум: создание XML-документа в VS



Теперь можно приступить к созданию XML-документов. Выполнение большей части черной работы VS берет на себя, и мы даже сможем создать XSD-схему для нашего XML-документа, не написав ни единой строки кода!

1. Откройте VS и выберите пункт меню **File | New | File...** (при этом не требуется, чтобы уже был открыт какой-либо проект). На экране появится окно, представленное на рисунке слева.

2. В меню New File выберите пункт XML File и щелкните мышью на Open. VS создаст новый XML-документ. Обратите внимание на то, что в XML-декларацию, которая используется VS, включен атрибут `encoding` (кроме того, VS выводит атрибуты и элементы разными цветами, однако это не слишком хорошо заметно при черно-белой печати):

```
<?xml version="1.0" encoding="utf-8" ?>
```

3. Сохраните файл, нажав клавиши *Ctrl + S*, или выбрав пункт меню File | Save XMLFile1. VS запросит вас, куда следует записать данный файл и какое имя ему присвоить; сохраните его в папке `BegCSharp\Chapter18` под именем `GhostStories.xml`.

4. Переведите курсор на следующую после XML-декларации строку и введите текст `<stories>`. Обратите внимание на то, что VS автоматически добавляет закрывающий тег непосредственно после того, как мы вводим знак "больше чем", которым оканчивается открывающий тег:

```
<?xml version="1.0" encoding="utf-8" ?>
<stories></stories>
```

5. Введите следующий XML-текст:

```
<?xml version="1.0" encoding="utf-8" ?>
<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>
```

Data for story

	title	rating
▶ ⊞	A House in Aungier Street	eerie
⊞	The Signalman	atmospheric
⊞	The Turn of the Screw	a bit dull
*		

6. Щелкните правой кнопкой мыши на XML-тексте в окне, в котором выведен код, и выберите пункт **View Data** из раскрывшегося меню. Это приведет к тому, что VS выведет данные, содержащиеся в XML-файле в табличной форме, как если бы они были считаны из реляционной базы данных (см. рис. слева).

7. Мы располагаем возможностью редактировать данные, представленные в таблице, то есть вносить изменения в XML-документ прямо с помощью этой таблицы без необходимости набирать теги. Щелкните мышью на клетке в столбце `title` в пустой строке в нижней части сетки и введите текст `"Number 13"`. Теперь переместитесь на соседнюю клетку в столбце `rating` и введите `"mysterious"` (см. рис. справа). Таким образом мы ввели новый рассказ, но нам необходимо ввести еще имя его автора. Для этого щелкните мышью на значке плюс, расположенном рядом с новой строкой. Это приведет к появлению ссылки на элемент `<author>`.

Data for story

	title	rating
⊞	A House in Aungier Street	eerie
⊞	The Signalman	atmospheric
⊞	The Turn of the Screw	a bit dull
▶ ⊟	Number 13	mysterious
	story_author	
*		

Data for story

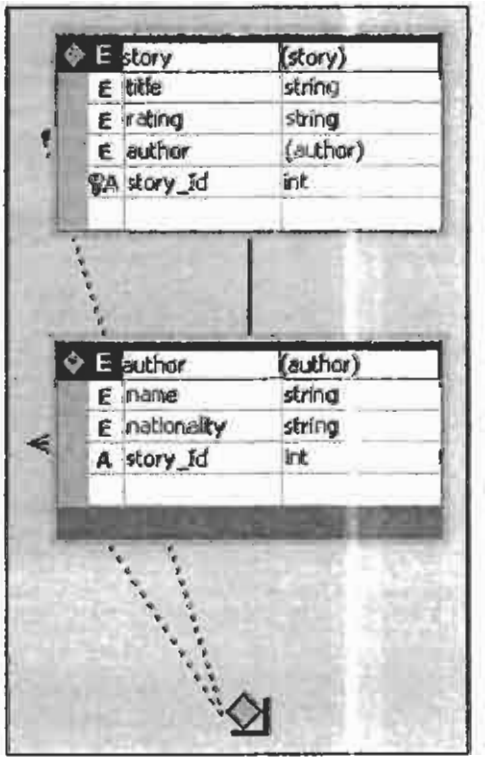
story: title: Number 13		rating: mysterious
	name	nationality
▶	MR James	English
*		

8. Щелкните мышью на этой ссылке, и вы увидите другую таблицу, в которой есть возможность задать имя и национальность автора (см. рис. слева). Введите в соответствующие столбцы `MR James` и `English` (не забудьте нажать *Enter* после того, как вы ввели национальность, иначе все данные будут потеряны).

9. Теперь щелкните правой кнопкой мыши на сетке с данными и выберите пункт **XML Source**. В результате должен быть добавлен новый элемент, расположенный непосредственно перед закрывающим тегом `</stories>`:

```
<story>
   <title>Number 13</title>
   <rating>mysterious</rating>
   <author>
      <name>MR James</name>
      <nationality>English</nationality>
   </author>
</story>
```

10. Завершающим действием мы заставим VS сгенерировать XSD-схему для созданного XML-документа. Щелкните правой кнопкой мыши на сетке с данными и выберите пункт **Create Schema**. VS создаст не только XSD-схему, но и диаграмму для наглядного представления этой схемы (см. рис. справа).



Обратите внимание на то, как VS добавил поле `story_Id`, *идентифицирующее каждый элемент* `<story>`. *Это оказывается полезным, если возникает необходимость преобразовать XML-документ в реляционные данные, поскольку базе данных потребуется способ, который позволяет определять, кто является автором каждого рассказа,— в XML-документе мы просто располагаем элемент* `<author>` *непосредственно под элементом* `<story>`.

11. Для то чтобы просмотреть полученную схему, щелкните правой кнопкой мыши на диаграмме и выберите пункт View XML Source.

Автодокументирование на XML в C#

До настоящего момента мы рассматривали XML совершенно абстрактно, и вы, возможно, испытывали чувство некоторого удивления — какое это все имеет отношение к C#? Зачем нам может понадобиться XML? В оставшихся главах данной книги мы рассмотрим некоторые варианты практического использования XML, однако существует одно применение XML, которое весьма специфично именно для C#. VS позволяет включать в файлы с программами на C# комментарии особого вида, которые затем могут быть извлечены и откомпилированы в отдельный файл. Такой файл представляет собой некоторый XML-документ. Это означает, что XML-анализаторы (например, встроенные в VS.NET) могут с легкостью извлекать информацию из такого файла. VS подготавливает эту информацию для вывода ее в виде web-страниц; в результате мы получаем прекрасно выглядящую документацию с помощью всего нескольких комментариев, включенных в код!

Часть этих специальных комментариев VS включает в программу автоматически. Они начинаются с трех прямых слэшей (///), а не с двух, как обычные комментарии. Например, VS добавляет эти строки перед описанием класса `Class1`, который автоматически включается VS в наш проект по созданию Windows- или консольного приложения:

```
/// <summary>
/// Summary description for Class1 /Общее описание класса Class1/
/// </summary>
```

Теперь можно сразу распознать в `<summary>` и `</summary>` открывающий и закрывающий теги XML-элемента. Для того чтобы увидеть, каким образом можно перейти от этого простого элемента к автоматическому созданию документации, добавим некоторые комментарии в приложение Мандельброта, созданное нами в главе 4.

Практикум: создание документации для класса

1. Откройте проект `Ch04Ex06` из главы 4.

2. Начните с того, что добавьте что-либо более полезное в элемент `<summary>`, который VS создал для класса `Class1`. Замените текст, расположенный между двумя тегами, на следующий (строка должна начинаться с трех прямых слэшей):

```
/// <summary>
/// Этот класс создает множества Мандельброта в консольном окне!
/// </summary>
```

```
class Class1
{
    /// <summary>
    /// |
    /// </summary>
    /// <param name="args"></param>
    static void Main(string[] args)
    {
```

3. Теперь создадим некоторую документацию для метода `Main()`. Добавьте пустую строку перед началом определения метода и введите три слэша (см. рис. слева). Смотрите, что произойдет после этого!

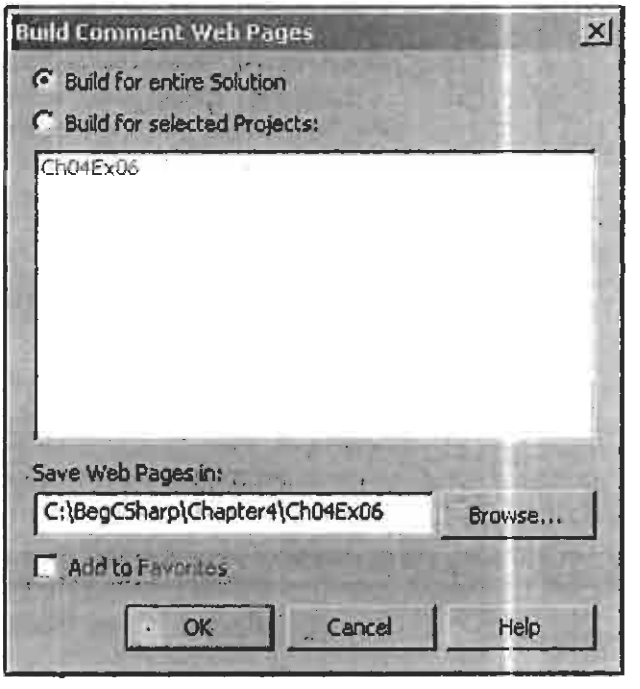
VS добавляет XML-теги для элемента `<summary>` (такого же, как тот, что использовался в классе `Class1`), однако кроме того, VS просматривает следующую строку, распознает в ней описание метода и выделяет его параметры. После этого для каждого параметра создается элемент `<param>` (в данном случае у нас есть единственный параметр — строковый массив с именем `args`, который может использоваться для считывания данных из командной строки в процессе выполнения приложения). Имя параметра при этом помещается в атрибут `name`.

4. Добавьте некоторое описание в элементы `<summary>` и `<param>`:

```
/// <summary>
/// Это метод Main() для класса Class1 –
/// именно здесь нами вызывается генератор Мандельброта!
/// </summary>
/// <param name="args">
/// Параметр args используется для считывания аргументов,
/// передаваемых из окна консоли.
/// </param>
```

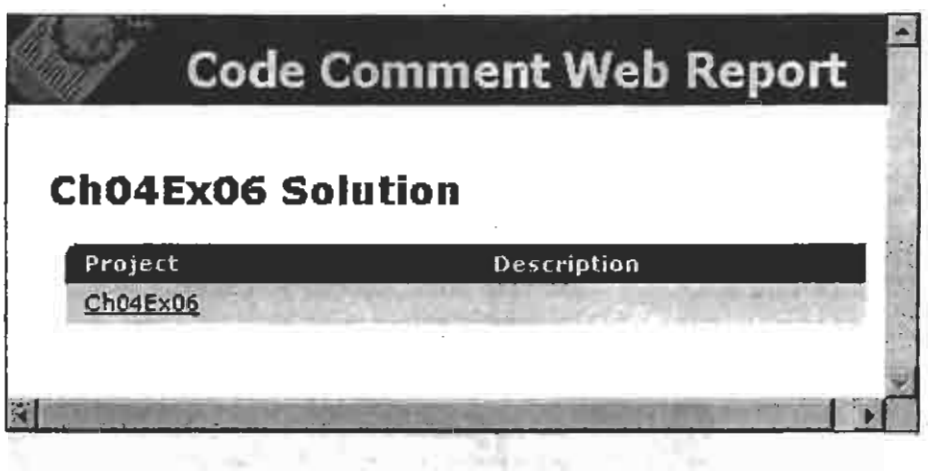
Обратите внимание на то, что VS знает, что вы находитесь в процессе набора комментариев для автодокументирования, и когда вы достигаете конца строки и нажимаете *Enter*, происходит автоматическое добавление трех слэшей на новой строке!

5. Теперь попытаемся преобразовать все эти комментарии в документацию. Выберите пункт меню **Tools | Build Comment Web Pages...**, откроется диалоговое окно, представленное на рисунке справа.

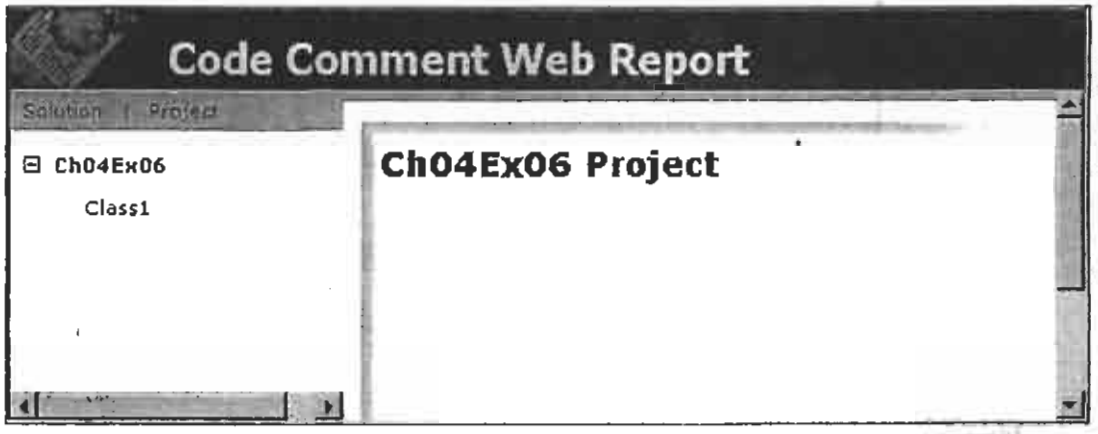


Если наше VS решение состоит более чем из одного проекта, можно определить здесь, следует ли создавать документацию для всего решения или только для отдельных проектов. Поскольку данное решение включает в себя только один проект, то оставьте выбранной опцию **Build for Entire Solution** (построение для всего решения). Можно также указать директорию, в которой будут сохраняться сгенерированные web-страницы.

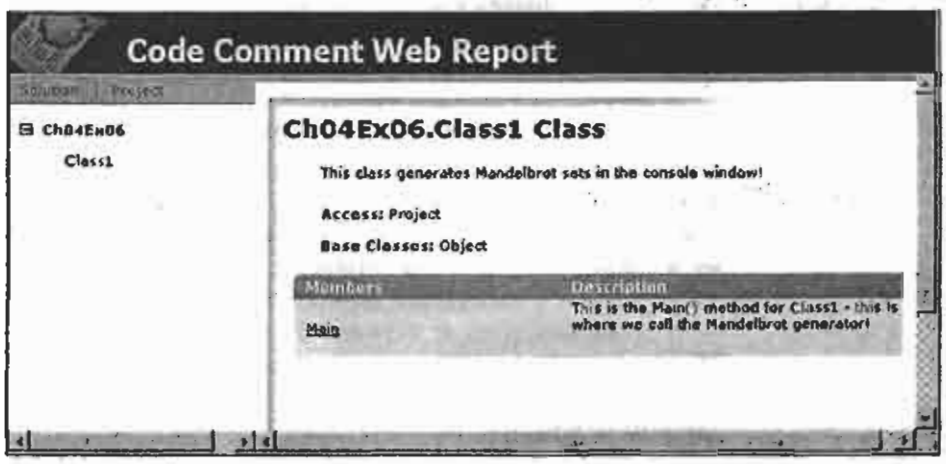
6. Щелкните мышью на **OK**, заставив тем самым VS сгенерировать web-страницы. Internet Explorer входит в состав VS, поэтому когда мы нажимаем **OK**, VS автоматически выводит на экран основную страницу документации для данного решения (см. след. рис.).



7. Щелкните мышью на ссылке `Ch04Ex06` — это приведет к открытию страницы документации для нашего проекта (единственного для данного решения):

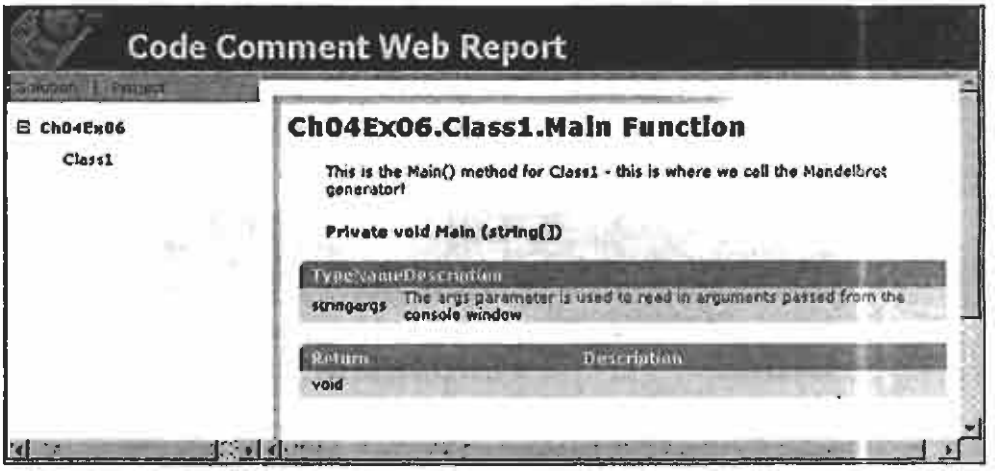


8. Откройте документацию для класса `Class1`, щелкнув мышью на значке "плюс", расположенном рядом со ссылкой на `Ch04Ex06`, а затем щелкните мышью на ссылке на `Class1`, которая появится ниже:



Здесь можно увидеть страницы, сгенерированные на основе наших комментариев. Страница документации, посвященная нашему классу, содержит введенное общее описание класса, список членов класса с общим описанием для каждого из них. В ней также содержится некоторая дополнительная информация, используемая VS для собственных нужд, — полное имя класса, включая пространство имен, уровень доступа к классу, а также класс, от которого данный класс происходит.

9. В завершение щелкните мышью на ссылке `Main`, для того чтобы просмотреть документацию по методу `Main()`. На этой странице выводятся описание метода и его параметра, а также его сигнатура:



К сожалению, эта страница не слишком хорошо отформатирована — описание параметра `args` выглядит не очень красиво.

Итоги

Visual Studio.NET предоставляет целый спектр инструментов, предназначенных для быстрого создания приложений, которые позволяют работать с базами данных. Самое замечательное — для этого не приходится писать почти никакого кода. На протяжении данной главы мы рассмотрели несколько механизмов вывода данных. Начиная с основополагающего текстового окна и заканчивая более сложной сеткой данных,— все они основываются на мощных возможностях и гибкости ADO.NET и объекта `BindingContext`, имеющегося в каждой форме Windows.

В настоящей главе мы рассмотрели вопрос о том, каким образом эти инструменты могут использоваться для установления соединений с источниками данных и для создания приложений для работы с данными, при этом требуется написать лишь незначительное количество строк кода. Однако автоматически сгенерированный код никогда не бывает столь же эффективен, как написанный вручную, поэтому в следующей главе мы познакомимся с тем, как можно развить то, что мы изучали здесь, и перейдем к самостоятельному написанию кода в ADO.NET.

В последнем разделе настоящей главы мы перешли к обсуждению Extensible Markup Language (XML — расширяемый язык разметки) — текстового формата, предназначенного для хранения и извлечения данных. Мы рассмотрели правила, которым необходимо следовать, для того чтобы создаваемые XML-документы были

правильно оформленными, а также то, каким образом мы можем проверять их допустимость, основываясь на XSD и XDR-схемах. Здесь мы также столкнулись с тем, что VS выполняет за нас всю черную работу и даже может создавать XSD-схемы на основе наших XML-документов. В заключение мы познакомились с имеющейся в C# возможностью создавать документацию для наших программ посредством использования специальных комментариев в XML-формате. Если вы до сих пор не убедились в необходимости XML, то в следующей главе вы сможете узнать о том, насколько неразрывно связаны между собой XML и ADO.NET.

Упражнения

1. Внесите изменения в первый пример, включив в `sqlDataAdapter1` дополнительный столбец `ContactTitle`.
2. Добавьте в форму текстовое окно, в котором будет выводиться информация о `ContactTitle`.
3. Создайте новое приложение, которое будет выводить информацию, хранящуюся в таблице `products` базы данных `Northwind`.
4. Внесите изменения в код примера "`Перемещение по объекту DataSet`", которые позволят переходнть в начало объекта `DataSet`, если на последней записи нажимается кнопка **Next**, н в конец объекта `DataSet`, если на первой записи нажимается кнопка **Back**.
5. Создайте комплект web-страниц автодокументацин на XML для окончательной версии приложения `Ch18Ex01`.

19

Доступ к данным посредством ADO.NET

В предыдущей главе рассматривалось, каким образом мастера и генерирующие код инструменты, входящие в состав Visual C#, могут быть использованы при создании пользовательских интерфейсов и управляющих элементов для доступа к реляционным базам данных.

В настоящей главе будет рассматриваться реализация такого же доступа к данным с помощью кода, написанного непосредственно программистом, а не сгенерированного мастером. Для реалнзации такого подхода следует воспользоваться классами ADO.NET доступа к данным. Сначала сделаем небольшой обзор самого ADO.NET, затем перейдем к изучению структуры классов, входящих в ADO.NET, и, наконец, создадим несколько примеров программ, которые используют ADO.NET.

Что такое ADO.NET?

Под названием ADO.NET объединено множество классов, которые используются совместно с C# и Microsoft .NET Framework для осуществления доступа к данным, хранящимся в реляционном таблично-ориентированном формате. Сюда относятся реляционные БД Microsoft Access и SQL Server, а также некоторые и даже нереляционные источники данных. Система ADO.NET входит в состав .NET Framework и разработана специально для использования с языками программирования .NET, и прежде всего с C#.

ADO.NET находится в модуле `System.Data.dll`. (Более подробное описание названного модуля см. в главе 21.) В некотором смысле модуль `System.Data.dll` и есть ADO.NET, поскольку любой класс, содержащийся в этом модуле, по определению представляет собой составную часть ADO.NET. Сюда включены все классы пространства имен `System.Data` и один класс из пространства имен `System.Xml`. XML рассматривался в предыдущей главе, и в настоящей главе мы продолжим изучать поддержку, которую ADO.NET оказывает XML.

Откуда взялось название ADO.NET?

Возникает вопрос: откуда у этой составной частн .NET Framework взялось такое загадочное название — ADO.NET? Почему бы не назвать его просто `System.Data`? Названне ADO.NET пронсходит от названия ADO (ActiveX Data Objects, объекты данных ActiveX), широко используемого в предшествующих поколеннях технологий компании Microsoft множества классов, предназначенных для доступа к данным.

Имя ADO.NET используется, чтобы подчеркнуть, что данный интерфейс является наиболее предпочтительным при осуществлении доступа к данным в среде программирования .NET.

ADO.NET используется для тех же целей, что ADO, и предоставляет множество очень простых в использовании классов, предназначенных для осуществления доступа к данным, обновленным и расширенным для использования в среде программирования .NET. Хотя ADO.NET выполняет те же функции, что и ADO, однако классы, свойства и методы в ADO.NET существенно отличаются от используемых в ADO.

Цели, преследовавшиеся при создании ADO.NET

Эти цели включают: осуществление простого доступа к реляционным и нереляционным данным, унификация доступа к реляционным и XML-данным, поддержка многозвенных приложений в Интернете, а также возможность расширения, которая бы позволила поддерживать большее количество различных источников данных, чем допускалось предшествующими технологиями.

Простота доступа к реляционным данным

Основной целью при создании ADO.NET было обеспечение простого доступа к реляционным данным. Несложные в использовании классы представляют таблицы, столбцы и строки, применяемые в реляционных базах данных. Кроме того, в ADO.NET вводится класс `DataSet`, который представляет набор данных, извлеченных из связанных таблиц, в виде единого объекта, благодаря чему удается сохранить целостность структуры отношений между ними. Это новый подход, привнесенный ADO.NET, позволяет существенно расширить возможности интерфейса доступа к данным.

Расширяемость

ADO.NET является расширяемой системой — она представляет собой каркас, куда могут включаться различные провайдеры данных .NET (называемые также **управляемыми провайдерами**), которые могут создаваться для чтения и записи данных из самых разнообразных источников. ADO.NET поставляется с двумя встроенными провайдерами данных .NET, один из которых предназначен для источников данных OLE DB, а другой — для источников данных Microsoft SQL Server. OLE DB позволяет осуществлять доступ к данным, хранящимся в форматах Microsoft Access, базах данных сторонних производителей и нереляционных данных. Кроме того, компания Microsoft недавно сделала предварительное сообщение о провайдере данных ODBC.NET для ADO.NET, который призван существенно расширить возможности осуществления доступа из .NET к еще большему числу унаследованных форматов данных и базам данных, разработанных сторонними организациями.

Поддержка многозвенных приложений

ADO.NET рассчитана на поддержку многозвенных приложений. Названная архитектура является на сегодняшний день наиболее распространенной для бизнес-приложений и приложений электронной коммерции. Многозвенная архитектура предполагает выполнение различных логических частей приложения на нескольких серверах или в рамках нескольких различных процессов, каждая из которых называется звеном.

В ADO.NET для осуществления взаимодействия между звеньями используется открытый Интернет-стандарт XML-формата, который позволяет данным преодолевать защиту, используемую в Интернете, и допускает реализацию одного или нескольких звеньев, не подчиняющихся стандартам компании Microsoft.

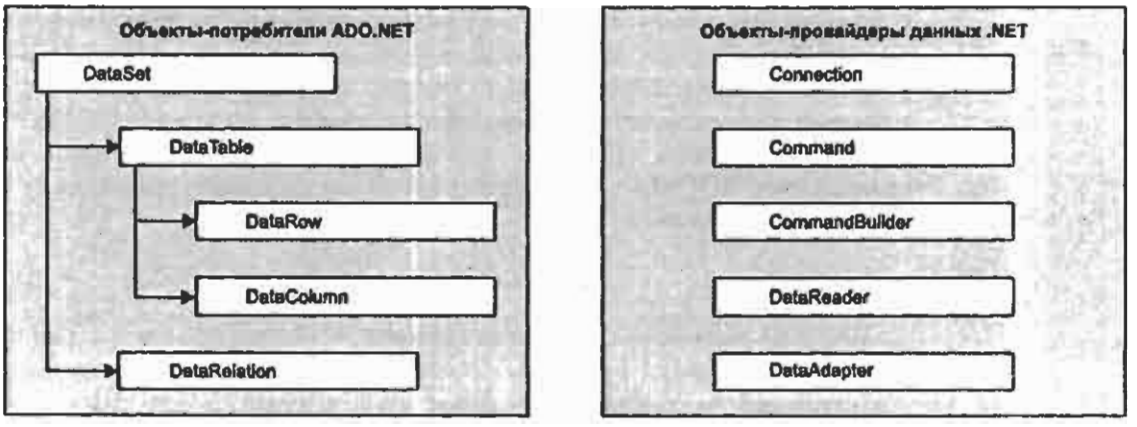
Унификация доступа к реляционным и XML-данным

Другое не менее важное предназначение ADO.NET — это обеспечение мостика между реляционными данными, хранящимися в виде строк и столбцов, и XML-документами, представляющими иерархическую структуру данных. .NET-технология построена на основе XML, и в ADO.NET она находит самое широкое применение.

Перейдем к непосредственному рассмотрению классов, из которых состоит ADO.NET.

Обзор классов и объектов ADO.NET

На изображенной ниже диаграмме представлены основные классы ADO.NET. Данная диаграмма не является диаграммой наследования, на ней всего лишь показаны отношения между наиболее широко используемыми классами:



Объекты-провайдеры специфичны для каждого конкретного типа источника данных; реальные считывание и запись данных из соответствующего источника осуществляются с помощью объектов-провайдеров, специфичных для данного источника. Объекты-потребители — это объекты, которые используемые для осуществления доступа и выполнения манипуляций над данными, после того как эти данные считаны в память. Для работы объектов-потребителей не требуется наличия установленного соединения; с данными, находящимися в памяти, можно работать, даже если соединения с базой данных более не существует. Для работы объектов-провайдеров активное соединение оказывается необходимым; прежде всего, эти объекты используются для считывания необходимых данных в память, затем — в зависимости от того, что требуется с ними сделать,— вы выполняете над ними соответствующие манипуляции в памяти с использованием объектов-потребителей и/или обновляете данные в источнике данных, используя объекты-провайдеры для записи изменений в источник данных.

Объекты-провайдеры

Эти объекты определяются для каждого провайдера данных .NET. Их имена предваряются некоторым именем, уникальным для данного провайдера; так, например, фактическое имя объекта, представляющего собой соединение, для провайдера OLE DB — `OleDbConnection`, для провайдера .NET для SQL Server используется класс `SqlConnection`, а при соединении, устанавливаемом провайдером .NET ODBC, класс носит имя `OdbcConnection`.

Объект Connection

Объект `Connection` — это первый из объектов, который вы будете использовать; необходимость в нем возникает до начала применения всех остальных объектов ADO.NET. Он обеспечивает установление основного соединения с вашим источником данных. Если используется база данных, которой требуется передавать имя пользователя и пароль, или база данных, расположенная на удаленном сетевом сервере, то все действия по установлению соединения с базой данных и регистрации в ней берет на себя объект `Connection`.

> *Если вы знакомы с классическим ADO, то вы, вероятно, обратили внимание на то, что объект `Connection` и другие объекты, выполняющие аналогичные функции в классическом ADO, в ADO.NET имеют аналогичные имена.*

Объект Command

Этот объект используется для передачи команды, такой как SQL-запрос, источнику данных, например, `"SELECT * FROM Customers"`, которая запрашивает данные, находящиеся в таблице `Customers`. Имена таких объектов специфичны для разных провайдеров; для SQL Server этот объект носит имя `SqlCommand`, а для OLE DB — `OleDbCommand`.

Объект CommandBuilder

Этот объект используется для создания SQL-комаид, которые позволяют модифицировать хранящиеся в объектах данные, полученные из одной таблицы. Этот объект будет рассматриваться более подробно в разделе, посвященном внесению изменений в данные. Среди названий этого объекта, имеющих отношение к конкретным провайдерам,— `SqlCommandBuilder` для SQL Server и `OleDbCommandBuilder` для OLE DB.

Объект DataReader

Это очень быстрый и простой в использованни объект, считываюшнй поток данных в режиме "только вперед" и "только чтение" (например, множество найденных заказчиков) из источнника данных. Этот объект обеспечивает максимальное быстродействие при простом считывании данных; образец использования этого объекта можно найти в первом из прнведенных ииже примеров. Среди специфичных для конкретных провайдеров названий этих объектов — `SqlDataReader` для SQL Server и `OleDbDataReader` для OLE DB.

Объект DataAdapter

Это многоцелевой класс, который выполняет различные операции, специфичные для конкретного источника данных, включая обновление измененных данных, наполнение наборов данных (см. ниже) и другие действия, которые встретятся нам в последующих примерах. Среди специфичных для конкретных провайдеров названий этих объектов — `SqlDataAdapter` для SQL Server и `OleDbDataAdapter` для OLE DB.

Объекты-потребители

Это объекты, использующиеся со стороны ADO.NET при разорванном соединении.

Объект DataSet

Этот объект представляет собой семейство связных таблиц, к которым приложение может обращаться как к единому целому. Например, таблицы `Customers`, `Orders` и `Products` могут находиться в одном и том же объекте `DataSet`, представляя каждого из клиентов и продукцию, которую данный клиент заказал в вашей компании. При помощи такого объекта можно быстро извлекать все необходимые данные из каждой таблицы, анализировать их и вносить соответствующие изменения, пока соединение с сервером не установлено, а затем сохранять все изменения на сервере посредством одной эффективной операции.

Объект `DataSet` обладает свойствами, которые позволяют осуществлять доступ к объектам более низкого уровня, представляющим отдельные таблицы, строки, столбцы и отношения. Вот эти объекты:

Объект DataTable

Представляет отдельные таблицы, хранящиеся в объекте `DataSet`, например, `Customers`, `Orders` или `Products`.

Объект DataColumn

Представляет какой-либо столбец в таблице, например, `CustomerName` или `OrderID`.

Объект DataRow

Представляет какую-либо связную строку данных из некоторой таблицы, например, `CustomerID`, имя, адрес и т. д. для какого-либо конкретного пользователя.

Объект DataRelation

Представляет отношение между двумя таблицами, устанавливаемое посредством общего столбца; так, напрнмер, в таблице `Orders` может присутствовать столбец `CustomerID`, позволяющий определить, кто из клиентов разместил данный заказ.

Помимо перечисленных здесь в ADO.NET существуют и некоторые другие объекты, однако на данный момент мы не будем углубляться в детали, а перейдем к рассмотрению некоторых примеров, демонстрирующих, как все это работает.

Использование пространства имен System.Data

Первым шагом на пути к использованию ADO.NET из кода на C# является ссылка на пространство имен `System.Data`, в котором находятся все классы ADO.NET. В начале каждой программы, в которой будет использоваться ADO.NET, необходимо использовать директиву `using` следующего вида:

```
using System.Data;
```

Провайдер данных .NET для SQL Server

Затем необходимо сослаться на провайдер данных .NET, применяемого для того источника данных, которым вы планируете воспользоваться. Если используется SQL Server (версии 7 или выше) либо MSDE, то наилучшее быстродействие и наиболее прямой доступ ко всем имеющимся возможностям осуществляется с помощью провайдера данных .NET, предназначенного для SQL Server, для обращения к которому используется следующая директива `using`:

```
using System.Data.SqlClient;
```

Провайдер данных .NET для OLE DB

Для большинства источников данных, отличных от SQL Server (Microsoft Access, Oracle и некоторых других), следует использовать провайдер данных .NET, предназначенный для OLE DB, для обращения к которому необходимо использовать директиву `using`:

```
using System.Data.OleDb;
```

Если есть провайдер данных .NET, специально предназначенный для используемой базы данных, то может появиться вполне законное желание воспользоваться именно этим провайдером, для чего придется применить соответствующую директиву `using`. Провайдер данных .NET для SQL Server представляет собой пример провайдера, очень чувствительного к типу используемого продукта.

Провайдер данных .NET для ODBC

Если вы работаете с каким-либо источником данных, для которого не существует "родного" провайдера или провайдера OLE DB (как, например, для PostgreSQL или некоторых других баз данных, созданных сторонними организациями), то наилучшей альтернативой является провайдер данных .NET для ODBC. Он представляет собой отдельный загрузочный модуль в Visual Studio.NET (находящийся по адресу http://msdn.microsoft.com/downloads/) — в .NET Framework следует найти ODBC .NET Data Provider Beta. После инсталляции загружаемого модуля он включается в состав модуля `System.Data.Odbc.dll`; при компиляции на этот модуль необходимо специально сослаться. Обратиться к провайдеру ODBC можно с помощью директивы `using`:

```
using System.Data.Odbc;
```

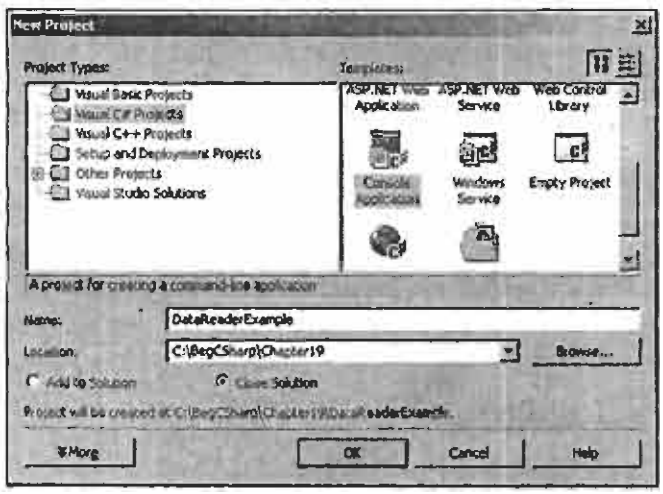
Практикум: простой пример осуществления доступа к данным, хранящимся в одной таблице

В этом примере мы попытаемся извлечь некоторые данные из одной таблицы, например, из таблицы `Customers`, используемой в качестве образца базы данных SQL Server/MSDE Northwind — той же самой базы данных, которую мы рассматривали в предыдущей главе.

Эта таблица состоит из строк и столбцов, в которых содержится информация о клиентах торговых представителей компании Northwind. В Visual Studio.NET эта таблица выглядит так, как показано на следующем рисунке.

Создадим с помощью ADO.NET программу, которая будет запрашивать столбцы данной таблицы с именами `CustomerID` и `CompanyName`.

CustomerID	CompanyName	ContactName	ContactTitle	Address	City
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim
BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg
BOLID	Bólido Comidas preparadas	Martín Sommer	Owner	C/ Araquil, 67	Madrid
BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	Marseille
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.	Tsawassen
BSBEV	B's Beverages	Victoria Ashworth	Sales Representative	Fauntleroy Circus	London
CACTU	Cactus Comidas para llevar	Patricio Simpson	Sales Agent	Cerrito 333	Buenos Aires
CENTC	Centro comercial Moctezuma	Francisco Chang	Marketing Manager	Sierras de Granada 9993	México D.F.
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Hauptstr. 29	Bern
COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate	Av. dos Lusíadas, 23	São Paulo



Данная программа, а также все последующие примеры в этой главе, будут создаваться одним и тем же способом — в виде проекта консольного приложения Visual Studio.NET; для этого и всех последующих примеров в настоящей главе следует использовать приведенную ниже последовательность шагов для компиляции и запуска приложения.

Запустите Visual Studio.NET и создайте новый проект C#. Для этого выберите пункт File | New | Project... основного меню, затем выберите шаблон Console Application из меню Visual C# Projects. Назовите вновь создаваемую программу DataReaderExample (см. рис. слева).

Переименуйте файл `Class1.cs` в `DataReaderExample.cs`, а затем полностью введите следующую программу C#, которая оказывается не слишком длинной:

```
using System;
using System.Data;                  // Использование пространства имен ADO.NET
using System.Data.SqlClient;        // Использование пространства
                                    // имен провайдера данных SQL Server

class DataReaderExample
{
   public static void Main()
   {
      // Задание строки соединения специально для SQL Server
      SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

      // Установка соединения
      thisConnection.Open();

      // Создание команды для данного соединения
      SqlCommand thisCommand = thisConnection.CreateCommand();

      // Задание SQL-запроса для этой команды
      thisCommand.CommandText =
            "SELECT CustomerID, CompanyName FROM Customers";

      // Выполнение объекта DataReader для заданной выше команды
      SqlDataReader thisReader = thisCommand.ExecuteReader();
```

```
            // Пока имеются непрочитанные строки
            while (thisReader.Read())
            {
                // Вывод столбцов с ID и именами
                Console.WriteLine("\t{0}\t{1}",
                thisReader["CustomerID"], thisReader["CompanyName"]);
            }

            // Закрытие объекта считывания
            thisReader.Close();

            // Закрытие соединения
            thisConnection.Close();
        }
    }
```

Откомпилируйте эту программу (*F7*) и запустите ее (*Ctrl-F5*). Вы увидите список идентификаторов клиентов и названий компаний, который приводится ниже:

```
ALFKI    Alfreds Futterkiste
ANATR    Ana Trujillo Empredados y helados
ANTON    Antonio Moreno Taqueria
AROUT    Around the Horn
BERGS    Berglunds snabbkop
BLAUS    Blauer See Delikatessen
. . .
```

Мы пройдем по каждой строке примера, для того чтобы понять, каким образом были получены такие результаты.

Как это работает

Первым шагом необходимо сослаться на пространство имен `System.Data` и на наш провайдер — так, как это описывалось ранее. В этих примерах мы будем использовать провайдер .NET для SQL Server, поэтому в начале программы должны быть помещены следующие строки:

```
using System.Data;
using System.Data.Sql;
```

Установление соединения с источником данных

Далее необходимо установить соединение со нашим источником данных. Это осуществляется посредством создания объекта соединения с использованием строки соединения. Строка соединения — это символьная строка, в которой содержится имя провайдера базы данных, с которой устанавливается соединение, необходимая регистрационная информация (пользователь базы данных, пароль и т. д.) и имя конкретной базы данных, с которой вы собираетесь работать. Мы подробно рассмотрим отдельные части этой строки соединения, однако не следует забывать о том, что эти строки для разных провайдеров существенно отличаются друг от друга, поэтому необходимо включать в них информацию, специфичную для используемого провайдера данных, если ваш провайдер отличается от рассматриваемого в настоящем примере. (Доступ к данным демонстрируется ниже в данной главе.)

Строка, в которой происходит создание объекта соединения, выглядит следующим образом:

```
SqlConnection thisConnection = new SqlConnection(
      @"Data Source=(local);Integrated Security=SSPI;" +
      "Initial Catalog=northwind");
```

SqlConnection — это имя объекта соединения провайдера данных .NET для SQL; если бы мы использовали OLE DB, то пришлось бы создавать объект с именем OleDbConnection. Строка соединения состоит из поименованных вхождений, отделенных одно от другого точкой с запятой. Первым является следующее:

```
Data Source=(local);
```

Это имя SQL Server, к которому осуществляется доступ. (local) — удобное сокращение имени SQL Server, обозначающего экземпляр сервера, который выполняется на вашей машине; это имя должно сработать прямо в таком виде.

Шаблоны .NET Framework SDK допускают опционную инсталляцию настольного средства SQL Server (называемого также MSDE) для выполнения шаблонов. По умолчанию этому экземпляру присваивается имя (local)\NetSDK. Следовательно, в качестве альтернативы можно попытаться использовать следующее имя сервера:

```
Data Source=(local)\NetSDK;
```

Обратите внимание на то, что знак @, расположенный в начале строки соединения, обозначает, что это — символьное значение, что позволяет использовать в строке обратный слэш; в противном случае, для того чтобы использовать символ обратного слэша в строке C#, необходимо применять escape-последовательность, состоящую из двух обратных слэшей (\\). Можно также указать фактическое имя компьютера:

```
Data Source=ROADRUNNER\NetSDK;
```

Если тот Server, с которым вы работаете, инсталлирован, то известно имя экземпляра SQL Server. В противном случае придется обратиться к вашему администратору SQL Server или сетевому администратору и узнать имя, которое следует использовать.

Следующая часть строки соединения определяет, как регистрироваться в базе данных; эта специальная фраза указывает на использование встроенной объединенной системы безопасности для регистрации в Windows и в SQL Server, поэтому не требуется задавать дополнительных имен и паролей:

```
Integrated Security=SSPI;
```

SSPI расшифровывается как **Security Support Provider Interface (интерфейс провайдера по обеспечению безопасности)**; это обозначает встроенную стандартную систему обеспечения безопасности для Windows и SQL Server. В завершение указывается конкретная база данных, которую мы собираемся использовать,— в данном случае это база данных Northwind:

```
Initial Catalog=northwind
```

Используемая в качестве образца база данных Northwind по умолчанию всегда присутствует в инсталляции SQL Server, хотя многие администраторы баз данных предпочитают опускать ее из соображений экономии места. Остальная часть программы работает в точности так же, как и в предшествующем примере, поэтому мы не будем повторяться.

Мы обладаем объектом соединения, настроенным на соответствующий компьютер и базу данных (это соединение пока не является активным; для активизации его необходимо открыть).

Метод Open

Как только появляется объект соединения, первое, что требуется сделать,— это открыть его, в результате будет установлено соединение с базой данных:

```
thisConnection.Open();
```

Если выполнение метода `Open()` завершится неудачей, например, не удалось обнаружить необходимый SQL Server, то будет сгенерирована исключительная ситуация `SqlException` и будет выведено следующее сообщение:

```
Unhandled Exception: System.Data.SqlClient.SqlException: Timeout expired.
(Необрабатываемая исключительная ситуация: System.Data.SqlClient.SqlException:
Истекло время ожидания.)
Possible reasons: timeout period elapsed prior to completion of the operation,
(Возможные причины: период ожидания истек раньше, чем была завершена операция,)
the server is not responding, or the maximum pool size was exceeded.
(сервер не отвечает, превышено максимальное значение пула.)
Please see the documentation for further details.
(За дополнительными подробностями обращайтесь к документации.)
at System.Data.SqlClient.SqlConnection.Open()
at DataReaderSqlExample.Main() in c:\begsharp\chapter19\datareaderexample\
datareaderexample.cs:line 15
```

Данное сообщение свидетельствует о том, что программа не может установить соединение с указанным SQL Server за приемлемое время. Убедитесь в том, что имя сервера в строке соединения не содержит ошибок.

Отправка запроса (SQL-команды)

На следующем этапе необходимо создать командный объект и передать ему SQL-команду для выполнения операции с базой данных (например, для извлечения некоторых данных). Вот код, который позволяет осуществить это:

```
SqlCommand thisCommand = thisConnection.CreateCommand();
thisCommand.CommandText = "SELECT CustomerID, CompanyName from Customers";
```

У объекта соединения имеется метод `CreateCommand()`, предназначенный для создания команды, привязанной к установленному соединению, именно это и используется для получения командного объекта. Сама команда присваивается свойству командного объекта `CommandText`. Нам требуется получить список идентификаторов клиентов и имен компаний, хранящихся в базе данных `Northwind`, в соответствии с этим создается команда SQL-запроса:

```
SELECT CustomerID, CompanyName from Customers
```

Команда `SELECT` — это SQL-команда, позволяющая извлекать данные из одной или нескольких таблиц. Весьма распространенными являются ошибки, заключающиеся в неправильном наборе имен таблиц, что приводит к возникновению другой исключительной ситуации:

```
thisCommand.CommandText = "SELECT CustomerID, CompanyName from Customer";
```

Стоп! В названии таблицы `Customers` пропущена буква `s`. Это приводит к возникновению следующей исключительной ситуации:

```
Unhandled Exception: System.Data.SqlClient.SqlException: Invalid object name 'Customer'...
(Необрабатываемая исключительная ситуация: System.Data.SqlClient.SqlException:
Неправильное имя объекта 'Customer'...)
```

Возможно, вы обратили внимание на то, что метод, сгенерировавший ошибку, называется `ExecuteReader()`; чтобы получить результаты, необходимо создать объект `DataReader` и выполнить команду.

Считывание результатов с помощью объекта DataReader

Объект DataReader — это простой, быстрый объект, предназначенный для немедленного получения результатов запроса. Он работает в режиме "только чтение", поэтому его нельзя использовать для внесения изменений в данные — к рассмотрению данного вопроса мы перейдем после этого примера. Как видно из предыдущего раздела, мы используем метод из последнего созданного нами объекта — командного объекта — для создания привязанного объекта, который потребуется нам в дальнейшем, в данном случае это объект DataReader:

```
SqlDataReader thisReader = thisCommand.ExecuteReader();
```

Метод ExecuteReader() осуществляет выполнение SQL-команды в базе данных, поэтому все ошибки базы данных генерируются именно на этом этапе; этот же метод создает объект для считывания полученных результатов. Именно этот объект вместе с содержащимися в нем результатами передается нашей программе; мы присваиваем его объекту thisReader.

Методы Read и Get

Есть несколько различных методов для извлечения результатов из объекта считывания. Метод Read() объекта DataReader производит считывание отдельной строки данных, полученных в результате выполнения запроса, и возвращает значение true, если там еще есть непрочитанные данные, и false в противном случае. Поэтому мы организовываем цикл while для считывания данных с помощью метода Read() и выводим результаты по мере их получения на каждой итерации:

```
while (thisReader.Read())
{
   Console.WriteLine("\t{0}\t{1}",
   thisReader["CustomerID"], thisReader["CompanyName"]);
}
```

Пока метод Read() возвращает значение true, функция Console.WriteLine("\t{0}\t{1}" выводит строки, каждая из которых состоит из двух элементов, разделенных символами табуляции (\t). Объект OleDbDataReader обладает свойством indexer (обсуждение индексаторов можно найти в главе 11). Индексатор позволяет обращаться к столбцам как к элементам массива по их именам: thisReader["CustomerID"], thisReader["CompanyName"]. В результате происходит вывод значений CustomerID и CompanyName:

```
ALFKI Alfreds Futterkiste
ANATR Ana Trujillo Empredados y helados
   ...
```

Когда метод Read() достигает последнего значения, он возвращает значение false и выполнение цикла while завершается.

Закрытие объектов Reader и Connection

Последним шагом является закрытие объектов, в число которых входят Reader и Connection. У каждого их них имеется свой собственный метод, к которым мы обращаемся, перед тем как закончить выполненне программы:

```
thisReader.Close();
thisConnection.Close();
```

Вот и все, что касается осуществления доступа к одной таблице!

Практикум: пример с использованием Microsoft Access

Эта же программа с незначительными изменениями может использоваться для доступа к той же БД в версии Microsoft Access (nwind.mdb). Она располагается в директории C:\Program Files\Microsoft Visual Studio Net\Common7\Tools\Bin, однако эту базу данных не следует использовать непосредственно — ее необходимо предварительно скопировать (например, во временную директорию C:\tmp\nwind.mdb), для того чтобы в любой момент вернуться к исходному варианту. Кроме того следует учитывать, что этот путь использовался в ранней версии Visual Studio, однако к моменту чтения данного раздела он мог измениться, в таком случае следует воспользоваться командой Search, если вы не смогли найти ее в указанном месте.

Все эти изменения — за исключением отдельных деталей строки соединения — будут работать для любых источников данных OLE DB. Рассмотрим пример. (Этот пример хранится под названием DataReaderAccessExample на web-сайте издательства Wrox.)

```
using System;
using System.Data;                  // Использование пространства имен ADO.NET
using System.Data.OleDb;            // Использование пространства имен
                                    // для провайдера данных OLE DB .NET

class DataReaderAccessExample
{
   public static void Main()        // все действия выполняются в функции Main()
   {
      // Создание объекта соединения для провайдера OLE DB Microsoft Access;
      // обратите внимание на символ @, предваряющий символьное значение
      // строки, который делает возможным использование обратных слэшей
      OleDbConnection thisConnection = new OleDbConnection(
            @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\tmp\nwind.mdb");

      // Открытие объекта соединения
      thisConnection.Open();

      // Создание объекта SQL-команды для данного соединения
      OleDbCommand thisCommand = thisConnection.CreateCommand();

      // Инициализация команды SQL SELECT, предназначенной
      // для извлечения необходимых данных
      thisCommand.CommandText =
            "SELECT CustomerID, CompanyName FROM Customers";

      // Создание объекта DataReader на основе предварительно
      // определенного командного объекта
      OleDbDataReader thisReader = thisCommand.ExecuteReader();

      while (thisReader.Read())
      {
         Console.WriteLine("\t{0}\t{1}", thisReader["CustomerID"],
         thisReader["CompanyName"]);
      }
      thisReader.Close();
      thisConnection.Close();
   }
}
```

Все сделанные изменения касаются только имен объектов и содержимого строки соединения.

Кроме того, директива `using`, определяющая провайдер данных, изменилась с

```
using System.Data.SqlClient;
```

на

```
using System.Data.OleDb;
```

Как это работает

Вместо объектов `SqlConnection`, `SqlCommand` и `SqlDataReader` мы создаем объекты `OleDbConnection`, `OleDbCommand` и `OleDbDataReader`. Эти объекты работают, в основном, так же, как и их аналоги для OLE DB; если вас интересуют тонкие отличия в их работе, то вы можете обратиться к документации по Visual Studio.NET.

Единственное отличие касается строки соединения, которую пришлось изменить полностью. Ее первая часть, состоящая из посылки `Provider`, определяет провайдера, используемого для данного типа БД. Для баз данных Microsoft Access этот провайдер всегда имеет имя ("Jet" — это имя процессора базы данных, включенного в Access):

```
Provider=Microsoft.Jet.OLEDB.4.0;
```

Если потребуется воспользоваться другим провайдером OLE DB для какой-либо базы данных или другого формата данных, то имя провайдера должно задаваться именно в предложении `Provider`.

Вторая часть строки соединения представляет собой предложение `Data Source`, и для OLE DB/Microsoft Access оно просто определяет имя файла, в котором находится база данных Microsoft Access (файл с расширением `.mdb`) и который необходимо открыть:

```
Data Source=C:\tmp\nwind.mdb
```

Обратите внимание на то, что знак @, предваряющий строку соединения в программе, определяет символьное значение строки в C#, что позволяет использовать в имени пути символ "обратный слэш"; в противном случае для задания имен файлов в C# потребовалось бы использовать escape-последовательность в виде двух обратных слэшей (\\).

Практикум: внесение изменений в данные

Теперь мы умеем считывать информацию из баз данных, но каким образом можно вносить изменения в эти данные? Приведем очень простой пример, в котором работа ведется с единственной таблицей, однако он позволит ввести несколько новых объектов, которые мы будем использовать в этой главе позднее.

Предположим, что один из наших клиентов — Bottom-Dollar Markets — изменил свое название на Acme, Inc. Требуется изменить название компании в базе данных. В этом примере мы снова воспользуемся базой данных `Northwind`, представленной в версии SQL Server/MSDE.

Полный исходный код представлен ниже (этот пример можно переписать — он хранится под именем `DataUpdateExample`):

```
using System;
using System.Data;                  // Использование пространства имен ADO.NET
using System.Data.SqlClient;        // Использование провайдера
                                    // данных .NET SQL Server


class DataUpdateExample
{
```

```
    public static void Main()
    {
        // Задание строки соединения специально для SQL Server
        SqlConnection thisConnection = new SqlConnection(
              @"Data Source=(local);Integrated Security=SSPI;" +
              "Initial Catalog=northwind");

        // Установка соединения
        thisConnection.Open();

        // Создание объекта DataAdapter для внесения
        // изменений и выполнения других операций
        SqlDataAdapter thisAdapter = new SqlDataAdapter(
              "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

        // Создание объекта CommandBuilder для формирования SQL-команд
        SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);

        // Создание объекта DataSet, предназначенного
        // для хранения связанных таблиц, строк и столбцов
        DataSet thisDataSet = new DataSet();

        // Наполнение объекта DataSet с помощью запроса,
        // определенного ранее для объекта DataAdapter
        thisAdapter.Fill(thisDataSet, "Customers");

        // Вывод данных до внесения изменений
        Console.WriteLine("name before change: {0}",
              thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);

        // Внесение изменений в данные: таблица Customers,
        // строка 9, столбец CompanyName
        thisDataSet.Tables["Customers"].Rows[9]["CompanyName"] = "Acme, Inc.";

        // Обращение к команде Update для внесения изменений
        // в соответствующую таблицу
        thisAdapter.Update(thisDataSet, "Customers");

        Console.WriteLine("name after change: {0}",
              thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);

        thisConnection.Close();
    }
}
```

В процессе выполнения программа выведет следующий выходной поток:

```
name before change: Bottom-Dollar Markets
(название до внесения изменений)
name before change: Acme, Inc.
(название после внесения изменений)
```

Теперь разберем детально исходную программу.

Как это работает

Первая часть программы аналогична предыдущему примеру; мы создаем объект соединения с помощью строки соединения и устанавливаем само соединение:

```
SqlConnection thisConnection = new SqlConnection(
      @"Data Source=(local);Integrated Security=SSPI;" +
      "Initial Catalog=northwind");
thisConnection.Open();
```

Затем мы создаем объект нового типа DataAdapter:

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
      "SELECT CustomerID, CompanyName FROM Customers", thisConnection);
```

Объект DataAdapter является гораздо более многоцелевым, чем объект DataReader; этот объект используется для самых разнообразных типов операций с данными, включая запросы, изменения и удаление. Мы инициализируем его с тем же самым SQL-запросом, что и командный объект в предшествующем примере.

Объект CommandBuilder

В простом случае внесения изменений в единственную таблицу не требуется знать, каким образом следует записывать SQL-команду, осуществляющую обновление данных; в этом случае можно воспользоваться удобным объектом CommandBuilder, который автоматически построит необходимые SQL-операторы и привяжет их к объекту DataAdapter:

```
SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);
```

Заметьте, что мы передаем объект thisAdapter конструктору объекта CommandBuilder в качестве аргумента. Далее происходит создание корректных SQL-команд и привязка их к переданному объекту DataAdapter в момент создания конструктором объекта CommandBuilder. Ниже в настоящей главе мы познакомимся и с другими SQL-операторами, а пока SQL выполняет всю работу за нас.

Объект DataSet

Теперь следует ввести объект, значимость которого будет становиться очевидной по мере дальнейшего обсуждения. Это объект DataSet:

```
DataSet thisDataSet = new DataSet();
```

Объект DataSet занимает центральное место в ADO.NET; он используется во всех операциях, независимо от их сложности. В DataSet содержится набор связанных объектов DataTable, представляющих таблицы базы данных, с которой ведется работа. У каждого объекта DataTable имеются дочерние объекты DataRow и DataColumn, которые соответственно представляют строки и столбцы таблицы из базы данных. Как вскоре будет видно, эти объекты позволяют осуществлять доступ ко всем отдельным элементам таблиц, строк и столбцов.

Метод Fill объекта DataAdapter

До того как использовать объект DataSet, необходимо наполнить его данными из базы данных. С этой целью используется метод Fill() объекта DataAdapter. Почему метод Fill() является методом объекта DataAdapter, а не объекта DataSet? Потому что DataSet является абстрактным представлением данных в памяти, в то время как DataAdapter является объектом, который осуществляет привязку объекта DataSet к конкретной базе данных. В качестве первого параметра методу Fill() передается объект, который должен быть наполнен, а в качестве второго параметра — имя объекта DataTable внутри объекта DataSet, в котором содержатся интересующие нас данные:

```
thisAdapter.Fill(thisDataSet, "Customers");
```

В данном случае требуется таблица Customers, поэтому мы обращаемся к соответствующему объекту DataTable в объекте DataSet с тем же именем. После того как объект DataSet оказывается наполненным, у нас появляется возможность обращаться к отдельным строкам и столбцам.

Осуществление доступа к таблицам, строкам и столбцам в объекте DataSet

Посмотрим, как выглядят данные до внесения в них изменений:

```
Console.WriteLine("name before change: {0}",
      thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);
```

Что здесь происходит? Мы выводим столбец CompanyName из строки с индексом, равным 9, из таблицы Customers, что приводит к получению выходного потока следующего вида:

```
name before change: Bottom-Dollar Markets
(название до внесения изменений)
```

Сложное выражение, начинающееся с thisDataSet имеет значение "Bottom-Dollar Markets". Рассмотрим детально, из чего оно состоит:

```
thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]
```

У объекта DataSet есть свойство Tables, в котором содержится семейство всех объектов DataTable. Это семейство представляет собой упорядоченный список, откуда следует, что имеется такая возможность доступа к каждому отдельному объекту DataTable, как если бы он являлся элементом массива, при этом названне таблицы используется в качестве индекса:

```
Tables["Customers"]
```

В рамках каждого объекта DataTable есть свойство Rows, которое представляет собой семейство отдельных объектов DataRow; они также хранятся в виде упорядоченного списка, а в качестве индекса используется их номер. Здесь задача немного упрощена, считается, что заранее известен индекс интересующей нас строки, значение которого равно 9 (и который в действительности обозначает десятую строку, поскольку отсчет индексов ведется с нуля; первая строка — это строка Rows[0] или строка с номером 0):

```
Rows[9]
```

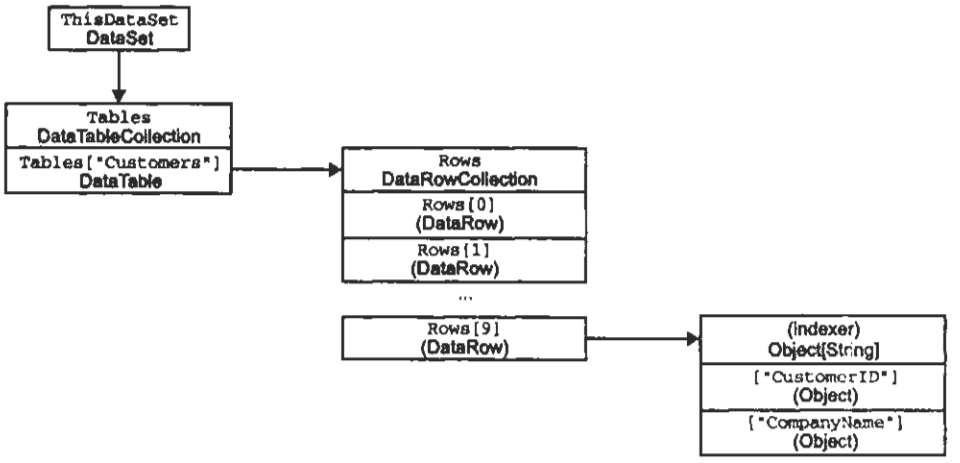
В отличие от данного примера, в реальной программе потребуется включать в SQL-запрос квалификатор, позволяющий выбирать интересующие нас строки,— вряд ли номера этих строк будут известны заранее. В следующем примере мы будем обсуждать вопрос о том, каким образом можно выбрать только те строки, которые нас интересуют.

Сам объект DataRow обладает индексируемым свойством, позволяющим получать доступ к отдельным столбцам по их названию. Мы используем этот индексатор для доступа к столбцу ["CompanyName"] строки Rows[9]: в таблице CompanyName.

Структура объекта DataSet, позволяющая получать доступ к таблицам, строкам и столбцам

Если только что описанная структура кажется слегка запутанной, давайте попробуем разобраться в ней с помощью рисунка на стр. 532.

Переменная thisDataSet обладает типом DataSet, определенным в момент ее объявления. У переменной DataSet есть свойство Tables типа DataTableCollection; отдельные члены названного семейства имеют тип DataTable; доступ к членам указанного семейства осуществляется по имени таблицы — в рассматриваемом случае нас интересует Tables["Customers"]. DataTable обладает свойством Rows



типа `DataRowCollection`; члены этого семейства имеют тип `DataRow`, а доступ к ним осуществляется по номеру строки.

Можно предположить, что `DataRow` обладает свойством типа `DataColumnCollection`, однако на самом деле все оказывается не так просто, поскольку доступ к отдельным столбцам каждой строки требуется осуществлять с учетом типа находящихся в нем данных, в результате чего символьная информация будет преобразовываться в строки, целочисленные значения — в объекты целого типа и т. д.

Для реализации такого подхода используется индексируемое свойство `DataRow`, которое представляет собой массив объектов общего типа (`System.Object`), индексируемый по имени столбца (которое представляет из себя строку). В этом случае, для получения доступа к данным с помощью индексатора, к объекту `DataRow` следует применить оператор массива. В нашем примере `Rows[9]` является объектом типа `DataRow`, поэтому выражение `Rows[9]["CompanyName"]` ссылается на объект, представляющий столбец с именем `CompanyName`, расположенный в строке с номером 9.

Еще один способ понять обсуждаемую структуру заключается в том, чтобы рассмотреть эквивалентную программу, в которой каждый элементарный объект выражения используется по отдельности:

```
// Пример с использованием нескольких объектов
DataTable customerTable = thisDataSet.Tables["Customers"];
DataRow rowNine = customerTable.Rows[9];
Object companyName = rowNine["CompanyName"];
Console.WriteLine("name before change: {0}", companyName);
```

В данном примере мы объявляем переменную `customerTable` типа `DataTable` и присваиваем ей таблицу `Customers`, извлекаемую из свойства `Tables` объекта `thisDataSet`. Затем мы объявляем переменную `rowNine` типа `DataRow` и присваиваем ей девятый элемент свойства `Rows` переменной `customerTable`. Наконец, мы объявляем переменную `CompanyName` типа `Object` и используем индексируемое свойство переменной `rowNine` для присваивания ей поля `CompanyName`.

Этот пример позволяет проследить всю цепочку последовательно используемых объектов, однако в программе проще использовать однострочное выражение, которое приводит к получению того же самого результата:

```
Console.WriteLine("name before change: {0}",
      thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);
```

Если вариант программы, в котором используется несколько объектов, является для вас более понятным, то вам всегда следует пользоваться только им. Для однократного обращения в полю — как в приведенном примере — подход с созданием переменных для каждого отдельного объекта и присваиванием им соответствующих значений может оказаться несколько неэффективным, если эти объекты будут использоваться неоднократно, подобный метод может оказаться более эффективным. Оптимизация, выполняемая компилятором, позволит компенсировать неэффективность как при одном подходе к программированию, так и при другом; отсюда следует, что наилучшим подходом к программированию является тот, который позволяет получить наиболее удобочитаемую программу.

Изменение данных, хранящихся в объекте DataSet

Для того чтобы изменить значение конкретного столбца данных, надо просто присвоить ему новое значение, как это делается в приведенной ниже строке программы, взятой из нашего примера:

```
thisDataSet.Tables["Customers"].Rows[9]["CompanyName"] = "Acme, Inc.";
```

Эта строка изменяет значение, хранящееся в столбце `CompanyName` в строке с номером девять в таблице `Customers`, на `"Acme, Inc."`. Однако эти изменения касаются только столбца объекта `DataSet`, хранящегося в памяти, а не собственно базы данных, расположенной на диске. Объекты `DataSet`, `DataTable`, `DataRow` и `DataColumn` являются хранящимся в памяти представлением данных, содержащихся в таблице. Для обновления самой базы данных необходимо вызвать метод `Update()`.

Метод Update() объекта DataAdapter

`Update()` — это метод объекта `DataAdapter`. Для обращения к этому методу необходимо указать объект `DataSet`, на основании которого будут вноситься изменения, и название таблицы, которой эти изменения касаются. Важно отметить, что имя объекта `DataTable` (`"Customers"`) совпадает с именем, которое использовалось при предшествующем вызове метода `Fill()`:

```
thisAdapter.Update(thisDataSet, "Customers");
```

Метод `Update()` автоматически осуществляет просмотр всех строк в таблице `DataTable` в поиске тех изменений, которые требуется внести в базу данных. У каждого объекта `DataRow`, принадлежащего семейству `Rows`, имеется свойство `RowState` (состояние строки), которое позволяет определять, была ли данная строка удалена, добавлена, изменена или она осталась без изменений. Все внесенные изменения находят свое отражение в базе данных.

Завершение работы

Теперь мы подтверждаем внесение изменений, выводя состояние данных "после" с помощью следующей записи:

```
Console.WriteLine("name after change: {0}",
      thisDataSet.Tables["Customers"].Rows[9]["CompanyName"]);
```

В заключение мы закрываем соединение:

```
thisConnection.Close();
```

Вот и все!

Практикум: добавление строк

В предыдущем примере мы внесли изменения в значения уже существующих строк, а как быть, если требуется внести совершенно новую строку или удалить уже существующую? Этого очень просто добиться с помощью методов объекта DataRow. Модифицируйте программу с тем, чтобы она создавала новую строку в таблице Customers, для чего замените код, расположенный между обращениями к методам Fill() и Update(), на код, выделенный серым цветом (пример называется DataAddRowExample):

```
thisAdapter.Fill(thisDataSet, "Customers");

Console.WriteLine("# rows before change: {0}",
      thisDataSet.Tables["Customers"].Rows.Count);

DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
thisRow["CustomerID"] = "ZACZI";
thisRow["CompanyName"] = "Zachary Zithers Ltd.";
thisDataSet.Tables["Customers"].Rows.Add(thisRow);

Console.WriteLine("# rows after change: {0}",
                  thisDataSet.Tables["Customers"].Rows.Count);

thisAdapter.Update(thisDataSet, "Customers");
```

В результате выполнения этой программы будет выведено следующее:

```
# rows before change: 91
(число строк до внесения изменений: 91 )
# rows after change: 92
( число строк после внесения изменений: 92)
```

Внимательно разберем эту программу.

Как это работает

Во-первых, чтобы представить картину "до", мы вводим новое свойство семейства Rows — Count. Оно используется в качестве счетчика общего количества строк, имеющихся в таблице:

```
Console.WriteLine("# rows before change: {0}",
                  thisDataSet.Tables["Customers"].Rows.Count);
```

Следующим действием мы создаем новый объект типа строки с помощью метода NewRow() объекта DataTable:

```
DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
```

Заметьте, что это приводит к созданию нового объекта типа строки, у которого имеются все те столбцы, из которых состоит таблица Customers, но при этом не происходит фактического добавления этой строки в объект DataSet. До того как у нас появится возможность осуществить это добавление, потребуется присвоить всем столбцам какие-нибудь значения:

```
thisRow["CustomerID"] = "ZACZI";
thisRow["CompanyName"] = "Zachary Zithers Ltd.";
```

Теперь мы можем на самом деле добавить новую строку с помощью метода Add() семейства Rows:

```
thisDataSet.Tables["Customers"].Rows.Add(thisRow);
```

Если снова обратиться к значению свойства Count после вызова метода Add(), то можно убедиться в том, что в действительности прибавилась одна строка:

```
Console.WriteLine("# rows after change: {0}",
                  thisDataSet.Tables["Customers"].Rows.Count);
```

Выходной поток свидетельствует о наличии 92 строк — на одну больше, чем было до внесения изменений. Как и в предшествующем примере для фактического добавления строки в базу данных, расположенную на диске, необходимо обратиться а методу Update():

```
thisAdapter.Update(thisDataSet, "Customers");
```

Не забудьте, что объект DataSet представляет собой копию данных, находящуюся в памяти и используемую без установления соединения; на самом деле соединение с базой данных, хранящейся на диске, осуществляется посредством объекта DataAdapter, и следовательно, для синхронизации данных, расположенных в памяти в объекте DataSet и находящихся на диске, требуется обратиться методу Update().

Если посмотреть на таблицу в VS после выполнения этой программы, то можно будет увидеть, что мы действительно успешно осуществили добавление строки:

CustomerID	CompanyName	ContactName	ContactTitle	Address	City
WARTH	Wartian Herkku	Pirkko Koskitalo	Accounting Manager	Torikatu 38	Oulu
WELLI	Wellington Importadora	Paula Parente	Sales Manager	Rua do Mercado, 12	Resende
WHITC	White Clover Markets	Karl Jablonski	Owner	305 - 14th Ave. S. Suite 38	Seattle
WILMK	Wilman Kala	Matti Karttunen	Owner/Marketing Assistant	Keskuskatu 45	Helsinki
WOLZA	Wolski Zajazd	Zbyszek Piestrzeniewicz	Owner	ul. Filtrowa 68	Warszawa
ZACZI	Zachary Zithers Ltd.	<NULL>	<NULL>	<NULL>	<NULL>

Обратите внимание на то, что в этой строке оказались заполненными только столбцы CustomerID и CompanyName, поскольку в нашей программе используются только эти столбцы. Остальные столбцы являются пустыми (на самом деле, с точки зрения SQL в них содержится значение NULL). Может показаться, что для заполнения, скажем, столбца Contact Name потребуется просто включить в программу следующую строку:

```
thisRow["ContactName"] = "Zylla Zithers";
```

Это, однако, не все, что потребуется сделать. При выполнении первоначального запроса мы создали объект DataSet, указав только два столбца — CustomerID и CompanyName:

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
      "SELECT CustomerID, CompanyName FROM Customers", thisConnection);
```

Обращение к столбцу ContactName приведет к возникновению ошибки, поскольку у построенного нами объекта DataSet такой столбец отсутствует. Можем исправить возникшую ситуацию, добавив столбец ContactName:

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
      "SELECT CustomerID, ContactName, CompanyName FROM Customers",
      thisConnection);
```

Либо можно выбрать все имеющиеся в таблице Customers столбцы с помощью следующей команды

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
      "SELECT * FROM Customers", thisConnection);
```

Звездочка (*) в SQL-команде SELECT используется для обозначения всех столбцов, имеющихся в таблице; внеся такое изменение вы получите возможность осуществлять доступ к значениям любых столбцов, хранящихся в БД. Однако следует отметить, что извлечение всех столбцов, если вы работаете с двумя или тремя из них, является неэффективным и этого следует избегать.

Практикум: осуществление поиска строк

Если бы вы попытались запустить программу из предыдущего примера более одного раза, то получили бы следующее сообщение:

```
Unhandled Exception: System.Data.SqlClient.SqlException:
(Необрабатываемая исключительная ситуация: System.Data.SqlClient.SqlException:)
Violation of PRIMARY KEY constraint 'PK_Customers'.
(Нарушение ограничений первичного ключа 'PK_Customers'.)
Cannot insert duplicate key in object 'Customers'.
(Не могу вставить повторяющийся ключ в объект 'Customers'.)
```

Это сообщение указывает на то, что выполнение функции `Add()` завершилось неудачей, так как ей потребовалось создать уже существующую строку. Определение таблицы `Customers` налагает требование уникальности значений, содержащихся в поле `CustomerID`, которое проверяется в момент присваивания соответствующему столбцу первичного ключа. В тот момент, когда вы попытались запустить программу во второй раз, значение `"ZACZI"` уже присутствовало в таблице, поскольку оно попало туда при первом запуске программы.

Внесем изменения в логику программы с тем, чтобы перед вставкой строки осуществлялся ее поиск. Семейство `Rows` объекта `DataTable` обладает методом `Find()` и оказывается чрезвычайно полезным в данной ситуации. Давайте изменим логику программы, которая окружает процедуру добавления строки, с тем чтобы вместо подсчета количества строк можно было использовать метод `Find()`. Ниже приводится полный исходный текст программы (среди программ, предназначенных для переписи, этот пример носит название `DataAddFindExample`); далее мы перейдем к детальному обсуждению этой программы:

```
using System;
using System.Data;                  // Использование пространства имен ADO.NET
using System.Data.SqlClient;        // Использование провайдера
                                    // данных .NET SQL Server

class DataAddFindExample
{
   public static void Main()
   {
      // Задание строки соединения специально для SQL Server
      SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

      // Установка соединения
      thisConnection.Open();

      // Создание объекта DataAdapter для внесения
      // изменений и выполнения других операций
      SqlDataAdapter thisAdapter = new SqlDataAdapter(
            "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

      // Создание объекта CommandBuilder для формирования SQL-команд
      SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);
```

```
            // Создание объекта DataSet, предназначенного для
            // хранения связанных таблиц, строк и столбцов
            DataSet thisDataSet = new DataSet();

            // Наполнение объекта DataSet с помощью запроса,
            // определенного ранее для объекта DataAdapter
            thisAdapter.Fill(thisDataSet, "Customers");

            Console.WriteLine("# rows before change: {0}",
                  thisDataSet.Tables["Customers"].Rows.Count);

            // Объявление объекта типа keys, предназначенного
            // для определения первичного ключа
            DataColumn[] keys = new DataColumn[1];
            keys[0] = thisDataSet.Tables["Customers"].Columns["CustomerID"];
            thisDataSet.Tables["Customers"].PrimaryKey = keys;

            DataRow findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI");

            if (findRow == null)
            {
               Console.WriteLine("ZACZI not found, will add to Customers table");

               DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
               thisRow["CustomerID"] = "ZACZI";
               thisRow["CompanyName"] = "Zachary Zithers Ltd.";
               thisDataSet.Tables["Customers"].Rows.Add(thisRow);
               if ((findRow =
                     thisDataSet.Tables["Customers"].Rows.Find("ZACZI")) != null)
               {
                  Console.WriteLine("ZACZI successfully added to Customers table");
               }
            }
            else
            {
               Console.WriteLine("ZACZI already present in database");
            }

            thisAdapter.Update(thisDataSet, "Customers");

            Console.WriteLine("# rows after change: {0}",
            thisDataSet.Tables["Customers"].Rows.Count);
            thisConnection.Close();
         }
      }
```

Как это работает

Начало данной программы вплоть до обращения к методу `Fill()` не отличается от программ, использовавшихся в предыдущих примерах. Мы используем свойство `Count` для вывода количества существующих в настоящий момент строк, а затем переходим к использованию метода `Find()`, позволяющего определить, не присутствует ли в таблице строка, которую мы собираемся добавлять.

До того как появится возможность обратиться к методу `Find()`, необходимо задать первичный ключ. Первичный ключ — это то, что будет использоваться для осуществления поиска; ключ состоит из одного или нескольких столбцов таблицы и содержит значение (или множество значений), которое идентифицирует данную строку таблицы уникальным образом, поэтому при осуществлении поиска по ключу можно найти одну и только одну строку. В БД `Northwind` для таблицы `Customers` в качестве первичного ключа используется ее столбец `CustomerID`:

```
DataColumn[] keys = new DataColumn[1];
keys[0] = thisDataSet.Tables["Customers"].Columns["CustomerID"];
thisDataSet.Tables["Customers"].PrimaryKey = keys;
```

Сначала мы создаем массив объектов DataColumn. Поскольку ключ может состоять из одного или нескольких столбцов, использование массива представляется наиболее естественной структурой; мы присваиваем массиву объектов DataColumn имя keys. Затем мы присваиваем первому элементу массива keys — keys[0] — значение столбца CustomerID из таблицы Customers. Наконец, мы присваиваем массив keys свойству PrimaryKey объекта DataTable таблицы Customers.

> *В качестве альтернативы существует возможность загружать информацию о первичных ключах непосредственно из БД, хотя по умолчанию этого не происходит. Тем не менее мы можем явно потребовать от ADO.NET загрузить информацию о первичном ключе, задав свойство MissingSchemaAction объекта DataAdapter до начала наполнения объекта DataSet следующим образом:*
>
> ```
> thisAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey
> thisAdapter.Fill(thisDataSet, "Customers");
> ```
>
> *Это позволяет реализовать ту же самую начальную установку первичного ключа посредством явной инициализации свойства PrimaryKey объекта DataAdapter.*

Как бы то ни было, но теперь мы готовы осуществить поиск строки!

```
DataRow findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI");
```

Метод Find() возвращает объект DataRow, поэтому мы создаем объект findRow типа DataRow для хранения результата поиска. Метод Find() принимает параметр, в котором содержится искомое значение; если ключ состоит из нескольких значений, это может быть массив объектов, однако в данном случае при использовании всего одного значения мы получаем возможность просто передать строку, имеющую значение "ZACZI", которое как раз и является тем значением CustomerID, поиск которого нами осуществляется.

Если методу Find() удается обнаружить соответствующую строку, то он возвращает объект DataRow, совпадающий с данной строкой; если методу Find() не удается обнаружить совпадение, то он возвращает ссылку на null, что мы и пытаемся определить:

```
if (findRow == null) {
   Console.WriteLine("ZACZI not found, will add to Customers table");

   DataRow thisRow = thisDataSet.Tables["Customers"].NewRow();
   thisRow["CustomerID"] = "ZACZI";
   thisRow["CompanyName"] = "Zachary Zithers Ltd.";
   thisDataSet.Tables["Customers"].Rows.Add(thisRow);
   if ((findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI")) != null)
   {
      Console.WriteLine("ZACZI successfully added to Customers table");
   }
}
else
{
   Console.WriteLine("ZACZI already present in database");
}
```

Если значение `findRow` равно `null`, то мы продолжаем выполнение программы и добавляем строку так, как мы это делали в предыдущем примере. Чтобы убедиться в том, что выполнение метода `Add()` завершилось успешно, мы вызываем метод `Find()` немедленно вслед за операцией добавления — просто для того, чтобы можно было быть уверенными в том, что все работает так, как надо.

Этот вариант программы с использованием метода `Find()` допускает многократное выполнение; сколько бы раз вы его ни запускали, он будет выполняться без ошибок. Однако эта программа не будет вызывать метод `Add()` в том случае, если строка с ключом `"ZACZI"` уже находится в базе данных. Теперь давайте научимся повторно выполнять и эту часть программы.

Практикум: удаление строк

Объект `DataRow` обладает методом `Delete()`, который позволяет удалять текущую строку. В новом варианте программы смысл оператора `if`, выполняемого над объектом `findRow`, изменен таким образом, что мы осуществляем проверку на предмет того, что значение `findRow` не равно `null` (другими словами, искомая строка найдена). После этого мы удаляем данную строку, вызывая метод `Delete()` на объект `findRow` (этот вариант программы называется `DataDeleteExample`):

```
using System;
using System.Data;                  // Использование пространства имен ADO.NET
using System.Data.SqlClient;        // Использование провайдера
                                    // данных .NET SQL Server

class DataDeleteExample
{
   public static void Main()
   {
      // Задание строки соединения специально для SQL Server
      SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

      // Установка соединения
      thisConnection.Open();

      // Создание объекта DataAdapter для внесения
      // изменений и выполнения других операций
      SqlDataAdapter thisAdapter = new SqlDataAdapter(
            "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

      // Создание объекта CommandBuilder для формирования SQL-команд
      SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);

      // Создание объекта DataSet, предназначениого для
      // хранения связанных таблиц, строк и столбцов
      DataSet thisDataSet = new DataSet();

      // Наполнение объекта DataSet с помощью запроса,
      // определенного ранее для объекта DataAdapter
      thisAdapter.Fill(thisDataSet, "Customers");

      Console.WriteLine("# rows before change: {0}",
            thisDataSet.Tables["Customers"].Rows.Count);

      // Объявление объекта типа keys, предназначенного для
      // определения первичного ключа
      DataColumn[] keys = new DataColumn[1];
      keys[0] = thisDataSet.Tables["Customers"].Columns["CustomerID"];
      thisDataSet.Tables["Customers"].PrimaryKey = keys;
```

```
        DataRow findRow = thisDataSet.Tables["Customers"].Rows.Find("ZACZI");

        if (findRow != null)
        {
            Console.WriteLine("ZACZI already in Customers table");
            Console.WriteLine("Removing ZACZI . . .");

            findRow.Delete();

            thisAdapter.Update(thisDataSet, "Customers");
        }

        Console.WriteLine("# rows after change: {0}",
            thisDataSet.Tables["Customers"].Rows.Count);

        thisConnection.Close();
    }
}
```

Как это работает

Обратите внимание на то, что вызов метода `Delete()` не приводит к удалению строки в базе данных, это происходит лишь в результате обращения к методу `Update()`, который фиксирует внесенные изменения.

Метод `Delete()` не осуществляет никакого удаления — он просто помечает строку как подлежащую удалению. Каждый объект `DataRow` в семействе `Rows` обладает свойством `RowState`, которое позволяет осуществлять контроль за тем, удалена ли, добавлена, модифицирована или оставлена без изменений данная строка. Метод `Delete()` просто присваивает свойству `RowState` значение `Deleted` (удалена), а затем метод `Update()` осуществляет фактическое удаление всех строк в семействе `Rows`, помеченных как `Deleted`.

В доступной в режиме on-line документации по Visual Studio.NET рекомендуется вызывать метод `AcceptChanges()` объекта `DataSet` после обращения к методу `Delete()`; это приведет лишь к удалению строки из объекта `DataSet` и не окажет никакого влияния на саму базу данных, поскольку метод `Update()` работает только с теми строками, которые он может обнаружить в семействе `Rows`, а отсутствующая там строка будет просто проигнорирована. Никогда не вызывайте метод `AcceptChanges()` до вызова метода `Update()`, если вам требуется удалить строку непосредственно из самой БД.

Все сказанное относится также и к методу `Remove()`, к которому следует обращаться, только если вам требуется удалить строки их семейства `Rows` объекта `DataSet`, но не непосредственно из самой базы данных.

Осуществление доступа к нескольким таблицам объекта DataSet

Одним из наиболее значимых преимуществ модели ADO.NET по сравнению с предыдущими моделями организации доступа к данным является то, что объект `DataSet` допускает одновременное хранение внутри себя сразу нескольких таблиц

и отношений между ними. Это означает, что появляется возможность передавать наборы связанных между собой данных между различными частями программы за одну операцию, а такой архитектуре оказывается присуще естественное поддержание целостности отношений между данными.

Объект `DataRelation` используется для описания отношений между различными объектами `DataTable`, хранящимися в `DataSet`. У каждого объекта `DataSet` в объектах `DataRelation` имеется семейство `Relations`, которое обеспечивает возможность осуществлять поиск и производить манипуляции над связанными таблицами.

Рассмотрим таблицы `Customers` и `Orders`. Поскольку каждый клиент может разместить сразу несколько заказов, то возникает вопрос, каким образом можно просмотреть все заказы, размещенные данным клиентом? Каждая строка таблицы `Orders` содержит поле `CustomerID` клиента, разместившего данный заказ, поэтому достаточно найти в этой таблице все строки, значение поля которых `CustomerID` совпадает со значением поля `CustomerID` данного клиента, хранящегося в таблице `Customers`:

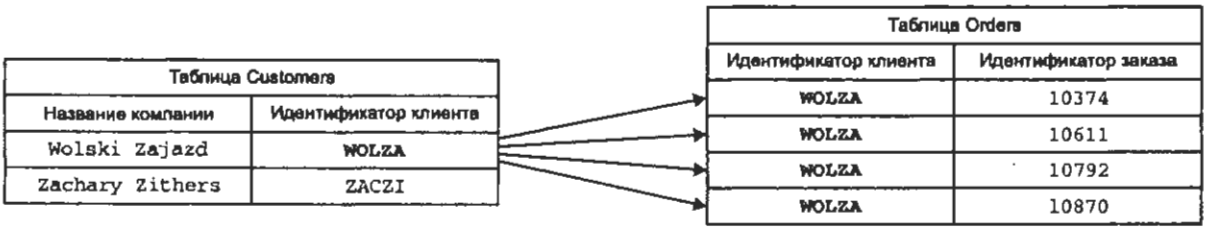


Таблица Customers	
Название компании	Идентификатор клиента
Wolski Zajazd	WOLZA
Zachary Zithers	ZACZI

Таблица Orders	
Идентификатор клиента	Идентификатор заказа
WOLZA	10374
WOLZA	10611
WOLZA	10792
WOLZA	10870

Совпадающие значения поля `CustomerID` в двух таблицах определяют отношение один-ко-многим между таблицами `Customers` и `Orders`. Есть возможность воспользоваться этим отношением в рамках ADO.NET, создав объект `DataRelation`, который будет представлять это отношение.

Поиск по отношениям

Чтобы воспользоваться отношениями, необходимо научиться переходить от строки в одной из таблиц к привязанным строкам в другой таблице. Этот процесс называется поиском по отношениям. Очень часто такой поиск заключается в переходе от родительской строки в одной таблице к привязанным дочерним строкам в другой. На диаграмме, приведенной выше, строка в таблице `Customers` может рассматриваться как родительская, а все привязанные к ней строки в таблице `Orders` могут считаться дочерними. Поиск может также происходить и в обратном направлении. Мы вернемся к этому понятию позже, когда подготовимся к тому, чтобы использовать его в своей программе. Сначала необходимо настроить нашу программу.

Практикум: пример использования объекта DataRelation

Ниже приводится полный листинг программы (`DataRelationExample`); после того как мы познакомимся со всей программой целиком, мы перейдем к обсуждению отдельных строк этой программы:

```
using System;
using System.Data;                  // Использование пространства имен ADO.NET
using Systen.Data.SqlClient;        // Использование провайдера данных OLE DB .NET

class DataAddFindExample
{
```

```
    public static void Main()
    {
        // Задание строки соединения специально для SQL Server
        SqlConnection thisConnection = new SqlConnection(
              @"Data Source=(local);Integrated Security=SSPI;" +
              "Initial Catalog=northwind");

        // Установка соединения
        thisConnection.Open();

        // Создание объекта DataAdapter для внесения
        // изменений и выполнения других операций
        SqlDataAdapter thisAdapter = new SqlDataAdapter(
              "SELECT CustomerID, CompanyName FROM Customers", thisConnection);

        // Создание объекта CommandBuilder для формирования SQL-команд
        SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);

        // Создание объекта DataSet, предназначенного для
        // хранения связанных таблиц, строк и столбцов
        DataSet thisDataSet = new DataSet();

        // Задание объектов DataAdapter для каждой из таблиц и их наполнение
        SqlDataAdapter custAdapter = new SqlDataAdapter(
              "SELECT * FROM Customers", thisConnection);
        SqlDataAdapter orderAdapter = new SqlDataAdapter(
              "SELECT * FROM Orders", thisConnection);
        custAdapter.Fill(thisDataSet, "Customers");
        orderAdapter.Fill(thisDataSet, "Orders");

        // Задание объекту DataRelation отношения между клиентами и заказами
        DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
              thisDataSet.Tables["Customers"].Columns["CustomerID"],
              thisDataSet.Tables["Orders"].Columns["CustomerID"]);

        // Вывод всех клиентов и идентификаторов их заказов
        foreach (DataRow custRow in thisDataSet.Tables["Customers"].Rows)
        {
           Console.WriteLine("Customer ID: " + custRow["CustomerID"] +
                             " Name: " + custRow["CompanyName"]);
           foreach (DataRow orderRow in custRow.GetChildRows(custOrderRel))
           {
              Console.WriteLine(" Order ID: " + orderRow["OrderID"]);
           }
        }
        thisConnection.Close();
    }
}
```

Как это работает

До создания объекта `DataRelation` необходимо создать объект `DataSet` и привязать к нему те таблицы из базы данных, которые мы собираемся использовать, как показано ниже:

```
DataSet thisDataSet = new DataSet();
SqlDataAdapter custAdapter = new SqlDataAdapter(
      "SELECT * FROM Customers", thisConnection);
SqlDataAdapter orderAdapter = new SqlDataAdapter(
      "SELECT * FROM Orders", thisConnection);
custAdapter.Fill(thisDataSet, "Customers");
orderAdapter.Fill(thisDataSet, "Orders");
```

Мы создаем объекты DataAdapter для каждой таблицы из тех, к которым мы собираемся обращаться. Затем наполняем объект DataSet данными из столбцов, с которыми мы планируем работать; в данном конкретном случае нам не приходится беспокоиться об эффективности, поэтому мы используем все существующие столбцы ("SELECT * FROM <таблица>").

Создание объекта DataRelation

Затем мы создаем объект DataRelation и привязываем его к объекту DataSet:

```
DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
      thisDataSet.Tables["Customers"].Columns["CustomerID"],
      thisDataSet.Tables["Orders"].Columns["CustomerID"]);
```

У объекта DataTable имеется свойство Relations, которое представляет собой семейство всех объектов DataRelation, описывающих отношения между таблицами, входящими в объект DataSet.

Для создания нового объекта DataRelation мы используем метод Add() семейства Relations, которому передается строка, представляющая собой название отношения (в данном примере — "CustOrders"), а также два объекта DataColumn: сначала родительский столбец CustomerID в таблице Customers, а затем дочерний столбец CustomerID в таблице Orders. Родительский столбец должен всегда предшествовать дочернему.

Использование метода GetChildRows

Теперь мы готовы к поиску клиентов и их заказов. Сначала следует организовать цикл foreach для вывода информации о каждом клиенте:

```
foreach (DataRow custRow in thisDataSet.Tables["Customers"].Rows)
{
   Console.WriteLine("Customer ID: " + custRow["CustomerID"] +
                     " Name: " + custRow["CompanyName"]);
```

Мы просто проходим в цикле по всему семейству Rows таблицы Customers и выводим значения CustomerID и CompanyName для каждого клиента. После того как выведена информация о конкретном клиенте, требуется вывести информацию о заказах, относящихся к данному клиенту.

Для этого мы используем вложенный цикл foreach, при инициализации которого вызывается метод GetChildRows() объекта DataRow. Мы передаем названному методу созданный нами объект DataRelation, а метод GetChildRows() возвращает нам объект DataRowCollection, в котором содержатся только те строки в таблице Orders, которые привязаны к данному клиенту:

```
   foreach (DataRow orderRow in custRow.GetChildRows(custOrderRel))
   {
      Console.WriteLine(" Order ID: " + orderRow["OrderID"]);
   }
}
```

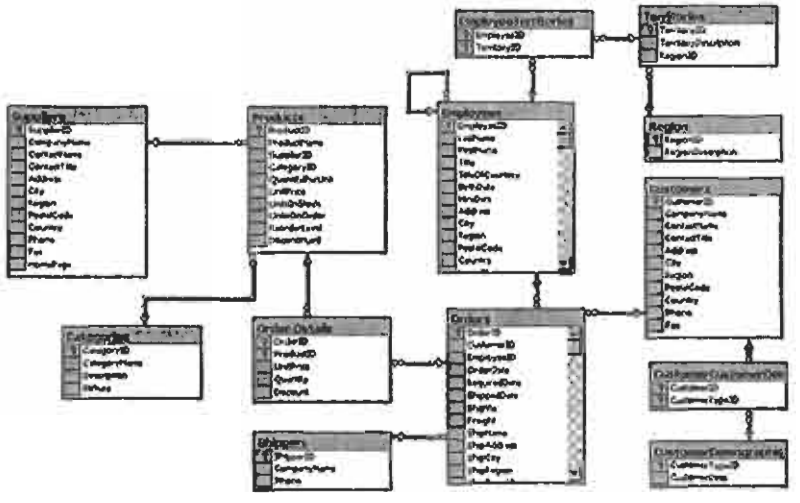
Теперь эта процедура будет повторяться для каждого клиента. Мы предваряем вывод значения OrderID несколькими пробелами, в результате чего заказы каждого клиента выводятся с определенным отступом после информации о самом клиенте:

```
...
Customer ID: WILMK Name: Wilman Kala
   Order ID: 10615
   Order ID: 10673
```

```
Customer ID: WOLZA Name: Wolski Zajazd
    Order ID: 10374
    Order ID: 10611
    Order ID: 10792
Customer ID: ZACZI Name: Zachary Zithers Ltd.
```

При таком выводе с отступами отношения родители-дети для каждого клиента и его заказов видны более четко. У клиента с идентификатором "Zachary Zithers Ltd." заказы отсутствуют, поскольку мы включили его в базу данных в одном из предыдущих примеров.

Практикум: работа с большим количеством таблиц



Расширим нашу программу, чтобы получить информацию о наименованиях, заказываемых каждым клиентом, и названиях продукции. Эта информация хранится в других таблицах базы данных `Northwind`. Для наглядности отношений между таблицами создадим диаграмму базы данных для БД `Northwind` в VS, где будут содержаться все таблицы,— так, как это проиллюстрировано на рисунке слева.

Если вы используете MSDE и отношения не были импортированы, то тогда все нормально — просто обратитесь к изображенной здесь диаграмме. Линии, проведенные между таблицами, обозначают отношения, при этом оба конца линии ведут к столбцу, обозначающему отношение. Отношение первичный ключ—внешний ключ изображено с помощью символа ключа у родительского столбца и значка бесконечности у дочернего столбца.

Мы собираемся выводить подробную информацию о каждом заказе клиента (включая наименования продукции), следуя отношениям, установленным между четырьмя таблицами, расположенными в самом верху диаграммы: `Customers`, `Orders`, `Order Details` и `Products`. В программе сначала инициализируется соединение, а затем создаются объекты `DataAdapter` для каждой из четырех таблиц:

```
DataSet thisDataSet = new DataSet();

SqlDataAdapter custAdapter = new SqlDataAdapter(
      "SELECT * FROM Customers", thisConnection);
custAdapter.Fill(thisDataSet, "Customers");

SqlDataAdapter orderAdapter = new SqlDataAdapter(
      "SELECT * FROM Orders", thisConnection);
orderAdapter.Fill(thisDataSet, "Orders");

SqlDataAdapter detailAdapter = new SqlDataAdapter(
      "SELECT * FROM Order Details", thisConnection);
detailAdapter.Fill(thisDataSet, "Order Details");

SqlDataAdapter prodAdapter = new SqlDataAdapter(
      "SELECT * FROM Products", thisConnection);
prodAdapter.Fill(thisDataSet, "Products");
```

Далее требуется построить объекты `DataRelation` для каждого отношения между четырьмя таблицами:

```
DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
      thisDataSet.Tables["Customers"].Columns["CustomerID"],
      thisDataSet.Tables["Orders"].Columns["CustomerID"]);

DataRelation orderDetailRel = thisDataSet.Relations.Add("OrderDetail",
      thisDataSet.Tables["Orders"].Columns["OrderID"],
      thisDataSet.Tables["Order Details"].Columns["OrderID"]);
```

Первое отношение в точности совпадает с отношением из предшествующего примера. Второе отношение — отношение между таблицами `Orders` и `OrderDetails`, причем в качестве связующего столбца используется столбец `OrderID`.

И последнее отношение — это отношение между таблицами `Order Details` и `Products`, в котором в качестве связующего столбца используется столбец с именем `ProductID`. Заметьте, что таблица `Products` является родительской (второй параметр), поскольку в отношении один-ко-многим она представляет сторону "один" (один и тот же продукт может фигурировать во многих заказах):

```
DataRelation orderProductRel = thisDataSet.Relations.Add("OrderProducts",
      thisDataSet.Tables["Products"].Columns["ProductID"],
      thisDataSet.Tables["Order Details"].Columns["ProductID"]);
```

После того как отношения определены, появляется возможность работать с ними. В данном случае в качестве основной структуры опять будут использоваться вложенные циклы `foreach`, на этот раз уровень вложенности будет равняться трем:

```
foreach (DataRow custRow in thisDataSet.Tables["Customers"].Rows)
{
   Console.WriteLine("Customer ID: " + custRow["CustomerID"];

   foreach (DataRow orderRow in custRow.GetChildRows(custOrderRel))
   {
      Console.WriteLine("\tOrder ID: " + orderRow["OrderID"]);
      Console.WriteLine("\t\tOrder date: " + orderRow["OrderDate"]);

      foreach (DataRow detailRow in orderRow.GetChildRows(orderDetailRel))
      {
         Console.WriteLine("\t\tProduct: " +
               detailRow.GetParentRow(orderProductRel)["ProductName"]);
         Console.WriteLine("\t\tQuantity: " + detailRow["Quantity"]);
      }
   }
}
```

Как и прежде, сначала мы выводим данные для родительской строки, а затем используем метод `GetChildRows()` для получения дочерних строк, привязанных к данной родительской строке. Внешний цикл совершенно не отличается от цикла из предыдущего примера. Далее мы выводим дополнительную информацию — дату заказа `Order Date` для данного `OrderID`, а затем получаем `OrderDetails` для этого же `OrderID`.

Внутренний цикл несколько отличается от предыдущих; для получения строки `Product`, мы обращаемся к методу `GetParentRow()`, который позволяет получить родительский объект, при этом движение осуществляется от стороны "многие" к стороне "один" в отношении "один-ко-многим". Иногда такой поиск в направлении от дочерних строк к родительским называется "восходящим" поиском, который противопоставляется нормальному поиску в направлении от родителей к детям

и называемому "нисходящим" поиском. Для осуществления восходящего поиска необходимо обратиться к методу `GetParentRow()`.

На этом пример программы завершен, не считая закрытия соединений, которое выполняется, как показано в предыдущих примерах. Если эту программу откомпилировать и выполнить, то можно увидеть следующий выходной поток (ниже приводится сокращенный вариант, представляющий собой последнюю часть вывода, который появится на вашем экране):

```
Customer ID: WOLZA
   Order ID: 10906
      Order Date 3/27/1996 12:00:00 AM
      Product: Sirop d'érable
      Quantity: 15
   Order ID: 10998
      Order Date 5/3/1996 12:00:00 AM
      Product: Guaraná Fantástica
      Quantity: 12
      Product: Sirop d'érable
      Quantity: 7
      Product: Longlife Tofu
      Quantity: 20
      Product: Rhönbräu Klosterbier
      Quantity: 30
   Order ID: 11044
      Order Date 5/23/1996 12:00:00 AM
      Product: Tarte au sucre
      Quantity: 12
Customer ID: ZACZI
```

Можно видеть все подробности, касающиеся заказов каждого клиента, при этом отступы используются для демонстрации иерархии родители—дети. У идентификатора клиента "ZACZI" по-прежнему отсутствуют какие-либо заказы, поскольку этот клиент был включен в базу данных в одном из предшествующих примеров.

Семейства ChildRelations и ParentRelations

У объекта `DataTable` есть свойство `ChildRelations`, где хранится семейство объектов `DataRelation` для отношений, в которых данная таблица задействована в качестве родителя. Это свойство может использоваться для обработки всех отношений таблицы:

```
foreach (DataRelation thisRelation in thisTable.ChildRelations) {
   foreach (DataRow thisParentRow in thisTable.Rows) {
      foreach (DataRow thisChildRow in
            thisParentRow.GetChildRows(thisRelation))
      {
         // . . . работа с дочерними строками . . .
      }
   }
}
```

У объекта `DataTable` кроме этого есть семейство `ParentRelations`, которое может быть использовано аналогичным образом. Нет совершенно никакой необходимости использовать эти семейства в нашем примере, поскольку в нем в каждый конкретный момент времени не приходится иметь дело с более чем одним родительским или дочерним отношением для каждой таблицы.

XML и ADO.NET

Поддержка XML — это одна из основных целей, которые преследовались при разработке ADO.NET, при этом внутреннее устройство ADO.NET также базируется на использовании XML. Поэтому нет ничего удивительного в том, что ADO.NET обладает самой разнообразной поддержкой XML, которая встроена в его объектную модель. Поскольку в предшествующей главе вы уже познакомились с XML, мы переходим к обсуждению того, каким образом он поддерживается в ADO.NET.

Поддержка XML в объектах DataSet ADO.NET

Поддержка XML в ADO.NET строится вокруг объекта `DataSet`, поскольку весь XML основан на отношениях и иерархически организованных данных. Объект `DataSet` обладает несколькими методами, которые позволяют работать с XML, и одним из наиболее простых в употреблении является метод `WriteXml()`, осуществляющий запись содержимого объекта `DataSet` в виде XML-документа.

Чтобы воспользоваться методом `WriteXml()`, следует просто сконструировать объект `DataSet` на основе имеющихся данных с помощью такого же кода, который использовался нами в предыдущих примерах,— использовав объект `DataAdapter` и метод `Fill()` для загрузки данных, описав отношения с помощью объектов `DataRelation` и т. д. После этого можно просто вызвать метод `WriteXml()` на созданный объект `DataSet`:

```
thisDataSet.WriteXml("nwinddata.xml");
```

Метод `WriteXml()` допускает осуществление записи в разные места, использованная в данном случае версия метода записывает XML просто в файл. Некоторая внешняя программа, для которой данный XML используется в качестве входных данных, получает возможность считать и обработать этот XML.

Существует также возможность использования метода `ReadXML()`, который позволяет считывать содержимое в виде XML и записывать его в объект `DataSet`.

Практикум: пример использования XML

Ниже приводится программа, в которой происходит создание объекта `DataSet` с последующим выводом его содержимого; эта программа очень похожа на программы, использованные в предшествующем примере, однако вместо сложных вложенных циклов `foreach` в ней используется единственное обращение к методу `WriteXml()`. Вы должны убедиться в наличии директории `C:\tmp`, до того как запускать эту программу.

```
using System;
using System.Data;
using System.Data.SqlClient;

class DataXmlExample
{
   public static void Main()
   {
      SqlConnection thisConnection = new SqlConnection(
            @"Data Source=(local);Integrated Security=SSPI;" +
            "Initial Catalog=northwind");

      thisConnection.Open();
```

```
        DataSet thisDataSet = new DataSet();

        SqlDataAdapter custAdapter = new SqlDataAdapter(
              "SELECT * FROM Customers", thisConnection);
        custAdapter.Fill(thisDataSet, "Customers");

        SqlDataAdapter orderAdapter = new SqlDataAdapter(
              "SELECT * FROM [Orders]", thisConnection);
        orderAdapter.Fill(thisDataSet, "Orders");

        SqlDataAdapter detailAdapter = new SqlDataAdapter(
              "SELECT * FROM Order Details", thisConnection);
        detailAdapter.Fill(thisDataSet, "Order Details");

        DataRelation custOrderRel = thisDataSet.Relations.Add("CustOrders",
              thisDataSet.Tables["Customers"].Columns["CustomerID"],
              thisDataSet.Tables["Orders"].Columns["CustomerID"]);
        custOrderRel.Nested = true;

        DataRelation orderDetailRel = thisDataSet.Relations.Add("OrderDetail",
              thisDataSet.Tables["Orders"].Columns["OrderID"],
              thisDataSet.Tables["Order Details"].Columns["OrderID"]);
        orderDetailRel.Nested = true;

        thisDataSet.WriteXml(@"c:\tmp\nwinddata.xml");
        Console.WriteLine(
              @"Successfully wrote XML output to file c:\tmp\nwinddata.xml");

        thisConnection.Close();
    }
}
```



Как это работает

Информацию о том, что содержимое заказов и сами заказы должны заноситься в XML с отступами, свойство `Nested` объектов `DataRelation` передает методу `WriteXml()`. В файле `nwinddata.xml` теперь хранятся все данные, находящиеся в таблицах (включая содержимое всех столбцов — поскольку при наполнении объекта `DataSet` мы воспользовались командой `SELECT * FROM`). Это вполне удобочитаемый и легко поддающийся анализу формат. Его можно просмотреть непосредственно в окне Microsoft Internet Explorer, показанном на рисунке слева.

Поддержка SQL в ADO.NET

В этой главе мы изучали основные операции ADO.NET, не имея никакого представления о языке SQL запросов к базам данных. Все команды ADO.NET, которые осуществляют чтение или запись информации в источники данных, транслируются в SQL-команды, непосредственно выполняющие операции с БД.

Практическое использование ADO.NET в реальных ситуациях потребует определенных знаний по SQL; для более полного знакомства с SQL следует обратиться к книгам *Beginning SQL Programming* или *Beginning SQL Server 2000 Programming*, которые выпущены издательством Wrox Press.

С некоторыми основными понятиями мы познакомимся прямо сейчас.

Использование SQL-команд в объектах DataAdapter

В приведенных ранее примерах мы использовали SQL-команду SELECT, которая возвращала все строки таблицы, например:

```
SqlDataAdapter thisAdapter = new SqlDataAdapter(
      "SELECT * FROM Customers", thisConnection);
```

Команда `SELECT` возвращает все строки и столбцы, имеющиеся в наличии в таблице клиентов, в момент вызова метода `Fill()` и загружает эту информацию в память программы. Это оказывается вполне подходящим для небольших таблиц, вроде таблицы `Customers` базы данных `Northwind`, которая состоит всего лишь из 11 столбцов и менее, чем из 100 строк данных, однако такой подход вряд ли будет нормально работать для больших таблиц, которые используются во многих бизнес-приложениях и состоят из сотен тысяч или даже миллионов строк.

Нужно строить команду `SELECT` таким образом, чтобы она возвращала только те данные, которые действительно требуется обрабатывать. Одним из способов является ограничение числа используемых столбцов в том случае, если ваша программа в действительности работает только с некоторыми из существующих столбцов, для чего в операторе `SELECT` необходимо задать только те столбцы, которые требуются на самом деле:

```
SELECT CustomerID, CompanyName FROM Customers
```

Этого следует избежать, если требуется добавлять строки, поскольку для этого следует задать значения для всех столбцов.

Использование ключевого слова WHERE с командой SELECT

Еще один способ минимизации количества данных, загружаемых в память, — это включение в SQL-оператор `SELECT` посылки `WHERE`, которая позволяет ограничить число выбираемых строк. Например, следующий оператор:

```
SELECT * FROM Customers WHERE CustomerID = 'ZACZI';
```

приведет к загрузке в память только одной строки, в которой имеется значение `ZACZI`, для чего будет использована малая часть памяти, которая потребовалась бы для загрузки всей таблицы целиком. В посылке `WHERE` также имеется возможность задавать диапазон интересующих нас значений, поэтому оператор следующего вида:

```
SELECT * FROM Orders WHERE OrderID BETWEEN 10900 AND 10999
```

загрузит только строки, у которых `OrderID` лежит внутри указанного диапазона.

Если существует возможность ограничить посредством использования посылки `WHERE` число строк, загружаемых из большой таблицы, то это всегда следует делать. Ни в коем случае не следует сначала загружать в объект `DataSet` все строки,

а затем осуществлять поиск с использованием цикла foreach; вместо этого для осуществления поиска такого рода следует использовать оператор SELECT с посылкой WHERE.

Вашей задачей является определение наиболее эффективного баланса между локальной обработкой данных на стороне клиента, на которой выполняется программа ADO.NET, и обработкой на стороне сервера, где выполняется SQL. Объектная модель ADO.NET и C# оказываются более приспособленными, чем SQL для выполнения сложных вычислений или поиска. Наполняйте объект DataSet данными, которые вам необходимо обработать, и выполняйте логику такого рода на стороне клиента. Помните, что ограничение количества выбираемых из каждой таблицы строк с помощью соответствующих условий позволит существенно увеличить быстродействие и сократить объем используемой памяти.

Рассмотрение SQL-команд SELECT, UPDATE, INSERT и DELETE

В SQL используется четыре основные команды для получения, изменения, добавления и удаления строк, составляющих таблицу: SELECT, UPDATE, INSERT и DELETE соответственно. В рассматривавшихся ранее примерах использовался объект CommandBuilder для создания SQL-команд, которые применялись для внесения изменений в базу данных:

```
SqlDataAdapter thisAdapter =
   new SqlDataAdapter("SELECT CustomerID from Customers", thisConnection);

SqlCommandBuilder thisBuilder = new SqlCommandBuilder(thisAdapter);
```

Этот объект генерирует SQL-команды, предназначенные для внесения изменений в данные (UPDATE, INSERT и DELETE), которые основываются на команде SELECT.

Практикум: пример, демонстрирующий SQL

В программе, которую мы сейчас создадим, можно увидеть команды, сгенерированные с помощью методов GetUpdateCommand(), GetInsertCommand() и GetDeleteCommand() объекта CommandBuilder:

```
using System;
using System.Data;
using System.Data.SqlClient;

class ShowSQLExample
{
   public static void Main()
   {
      OleDbConnection thisConnection = new OleDbConnection (
            "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB");
      thisConnection.Open();

      OleDbDataAdapter thisAdapter = new
         OleDbDataAdapter("SELECT CustomerID from Customers", thisConnection);

      OleDbCommandBuilder thisBuilder = new OleDbCommandBuilder(thisAdapter);
      Console.Writeline("SQL SELECT Command is: \n{0} \n",
                        this Adapter.SelectCommand.CommandText);

      OleCommand updateCommand = thisBuilder.GetUpdateCommand();
      Console.WriteLine("SQL UPDATECommand is:\n{0}\"n,
                        updateCommand.CommandText);
```

```
        OleDbCommand insertCommand = thisBuilder.GetInsertCommand();
        Console.WriteLine("SQL INSERT Command is:\n{0}\n",
                          insertCommand.CommandText);

        OleDbCommand deketeCommand = thisBuilder.GetDeleteCommand();
        Console.WriteLine("SQL DELETE Command is:\n{0}",
                          deleteCommand.CommandText);

        thisConnection.Close();
    }
}
```

Выходной поток этого примера будет иметь следующий вид:

```
SQL SELECT Command is:
SELECT CustomerID from Customers

SQL UPDATE Command is:
UPDATE 'Customers'SET 'CustomerID' =? WHERE ('CustomerID'=?)

SQL INSERT Command is:
INSERT INTO 'Customers'('CustomerID') VALUES(?)

SQL DELETE Command is:
DELETE FROM 'Customers'WHERE ('CustomerID'=?)
```

Как это работает

Обратите вниманне на то, что в командах `UPDATE` и `DELETE` используется посылка `WHERE`, сгенернрованная в объекте `CommandBuilder`.

Вопросительные знакн (?) обозначают параметры, которые системой выполнения ADO.NET будут заменяться в командах на реальные значения; например, если мы захотим использовать метод `Delete()` для удалення строки, содержащей идентификатор клиента `CustomerID` "ZACZI", то в момент вызова метода `Update()` для удаления строки `ZACZI` будет выполнена следующая команда:

```
DELETE FROM 'Customers'WHERE ('CustomerID'='ZACZI')
```

Чтобы вывести команду `SELECT`, мы воспользовались свойством `SelectCommand`, позволяющим получить команду непосредственно из объекта `DataAdapter`. У объекта `DataAdapter` также имеются свойства `UpdateCommand`, `InsertCommand` и `DeleteCommand`, которые позволяют напрямую считывать или записывать SQL-команды, использующиеся для внесения изменений. Разработчик, знакомый с SQL, имеет возможность оптимизировать эти команды, добиваясь лучшего выполнения по сравнению с командами, генерируемыми объектом `CommandBuilder`, особенно в тех случаях, когда SQL-оператором `SELECT` выбираются все имеющиеся столбцы.

Непосредственное выполнение SQL-команд

Если в программе требуется выполнить некоторую групповую операцию, такую как удаление или изменение всех строк, удовлетворяющих определенному условию, оказывается гораздо более эффективным (особенно при работе с большими таблицами) осуществлять это с помощью единой SQL-команды, нежели производить все необходимые вычисления в программе на C#.

В ADO.NET есть объекты `SqlCommand` и `OleDbCommand`, предназначенные для выполиения SQL-команд. У этих объектов есть метод `Execute()`, который осуществляет непосредственное выполнение SQL-команд. Нам уже приходилось пользоваться методом `ExecuteReader` в начале настоящей главы, когда мы изучали объект `DataReader`. Теперь мы приступаем к рассмотрению других методов `Execute()`.

Практикум: использование метода ExecuteScalar объекта SqlCommand

В качестве самого первого примера рассмотрим программу, которая подсчитывает число строк в таблице; эта программа аналогична примеру DataReader, однако в ней используется другой QL-оператор и SQL-метод Execute. В данном примере для выполнения запроса используется метод ExecuteScalar() объекта SqlCommand; этот метод используется для выполнения SQL-команд, которые могут возвращать только скалярные значения (т. е. одиночные значения — в отличие от методов, возвращающих сразу несколько строк, например, ExecuteReader()).

```
using System;
using System.Data;
using System.Data.SqlClient;

class DataExecuteScalarExample
{
   public static void Main()
   {
      SqlConnection thisConnection = new
      SqlConnection("Data Source=(local);" +
                    "Integrated Security=SSPI;Initial Catalog=northwind");
      thisConnection.Open();

      SqlCommand thisCommand = thisConnection.CreateCommand();
      thisCommand.CommandText = "SELECT COUNT (*) FROM Customers";
      Object countResult = thisCommand.ExecuteScalar();
      Console.WriteLine("Count of Customers = {0}", countResult);
      thisConnection.Close();
   }
}
```

Как это работает

В этой программе используется провайдер данных .NET SQL (System.Data.SqlClient). Общая структура программы не отличается от структуры самой первой программы настоящей главы — установления соединения с SQL Server на локальной машине с использованием системы обеспечения безопасности SSPI и базы данных Northwind.

Далее в программе создается объект SqlCommand, которому присваивается команда SELECT COUNT (*), представленная в текстовом виде. COUNT () — это SQL-функция, подсчитывающая число строк, которые удовлетворяют условию, заданному WHERE. После этого вызывается метод ExecuteScalar объекта SqlCommand, который выполняет запрос на получение счетчика строк. Значение счетчика выводится на экран, и программа завершает свою работу. Если выполнить эту программу для базы данных Northwind, то будет получен следующий выходной поток:

```
Count of Customers = 91
```

Это эквивалентно загрузке таблицы Customers в объект DataSet и вызову метода COUNT () на объект Rows(), что и делалось в предшествующих примерах. Почему следует отдать предпочтение выполнению этой операцию таким образом? На самом деле, все зависит от структуры данных и от того, каковы остальные действия, выполняемые в программе. Если вы оперируете небольшим объемом данных или загружаете все строки таблицы в объект DataSet по каким-либо другим причинам, то имеет прямой смысл воспользоваться DataTable.Rows.Count. Напротив, если вам требуется подсчитать точное количество строк (порядка миллиона) в очень большой таблице, то более эффективным будет выполнить запрос SELECT COUNT (*) с методом ExecuteScalar(), вместо того чтобы пытаться загрузить в память миллион строк.

Практикум: использование метода ExecuteNonQuery объекта SqlCommand

Рассмотрим еще один пример. Метод `ExecuteNonQuery()` объекта `SqlCommand` выполняет такие SQL-команды, которые вообще не возвращают никаких данных. SQL-операции, предназначенные для внесения изменений в данные, — `INSERT`, `UPDATE` и `DELETE`, не возвращают никаких данных; что на самом деле представляет интерес при выполнении этих команд — так это число обработанных строк, которое и возвращается данными методами.

Предположим, что один из наших поставщиков повысил на 5% цены на всю свою продукцию. Приведенная ниже программа демонстрирует, каким образом можно использовать объект `SqlCommand`, для того чтобы выполнить SQL-команду `UPDATE`, которая позволит повысить цены на 5% для всей продукции, поступающей от данного поставщика:

```
using System;
using System.Data;
using System.Data.SqlClient;

class DataSqlNonQueryExample
{
   public static void Main()
   {
      SqlConnection thisConnection = new SqlConnection(
            "Data Source=(local);" +
            "Integrated Security=SSPI;Initial Catalog=northwind");
      thisConnection.Open();

      SqlCommand thisCommand = thisConnection.CreateCommand();
      thisCommand.CommandText = "UPDATE Products SET " +
            "UnitPrice=UnitPrice*1.05 WHERE SupplierId=12";
      int rowsAffected = thisCommand.ExecuteNonQuery();
      Console.WriteLine("Rows Updated = {0}", rowsAffected);
      thisConnection.Close();
   }
}
```

Как это работает

Эта программа устанавливает соединение точно таким же образом, как и в предыдущих примерах. Затем в ней создается объект `SqlCommand`, которому присваивается команда `UPDATE`, представленная в текстовом виде. Затем происходит вызов метода `ExecuteNonQuery` объекта `SqlCommand`, который выполняет эту команду, возвращая число измененных в базе данных строк. Это число выводится на экран, после чего программа завершает свою работу. Если выполнить эту программу для базы данных `Northwind`, то будет получен следующий выходной поток:

```
Rows Updated = 5
```

Это указывает на то, что цены были изменены для пяти различных продуктов.

Дополнительная информация об ADO.NET

На этом мы завершаем введение в ADO.NET. Если вам захотелось узнать больше, то рекомендуем обратиться к документации Microsoft .NET Framework SDK, доступной в режиме on-line; поиск ADO.NET, осуществленный в режиме "Search Titles Only" (поиск только среди заголовков), позволит получить список обзоров по ADO.NET. Аналогичный поиск, выполненный для имен отдельных объектов,

таких как DataTable, позволит получить информацию о ссылках на этот конкретный объект, включая информацию о методах и свойствах этого объекта, которые мы не рассматриваем в настоящей книге по причине нехватки объема.

Итоги

Из этой главы мы узнали, что ADO.NET представляет собой составную часть .NET Framework, которая обеспечивает доступ к реляционным базам данных и другим источникам данных. ADO.NET разработана таким образом, чтобы обеспечить простую и гибкую схему для осуществления доступа к данным, она рассчитана на работу с приложениями, обладающими многозвенной архитектурой, и позволяет объединить реляционные данные и XML.

Классы ADO.NET хранятся в пространстве имен System.Data. Мы рассмотрели объектную модель ADO.NET и познакомились с предназначением основных ее объектов, включая объекты Connection, Command, DataReader, DataSet, DataTable, DataRow и DataColumn.

Мы также узнали, что провайдеры данных .NET позволяют осуществлять доступ к конкретным источникам данных и что ADO.NET может расширяться для работы с новыми источниками данных путем создания провайдеров данных .NET для источников. Мы изучили провайдеры данных .NET, включенные в состав ADO.NET для работы с источниками данных Microsoft SQL Server и OLE DB, а также узнали, что они находятся в пространствах имен System.Data.SqlClient и System.Data.OleDb соответственно.

Мы познакомились с тем, каким образом можно реализовать быстрый доступ к данным в режиме "только чтение" посредством объекта DataReader, а также с тем, как можно создавать эквивалентные программы для провайдеров данных .NET SqlConnection и OleDb. Мы узнали, каким образом можно модифицировать данные и добавлять новые строки с помощью объектов DataSet, DataAdapter и CommandBuilder. Мы также узнали, каким образом можно осуществлять поиск строк по первичному ключу и каким образом можно удалять строки.

Нами было рассмотрено, каким образом можно осуществлять доступ к нескольким таблицам, находящимся в DataSet, посредством объекта DataRelation, а также что нужно для того, чтобы создать XML-представление данных. В заключение мы коротко рассмотрели вопрос о том, каким образом можно воспользоваться преимуществами, предоставляемыми встроенной в ADO.NET поддержкой языка SQL для работы с базами данных, выключая вывод автоматически сгенерированных SQL-команд и непосредственное выполнение SQL.

Упражнения

1. Модифицируйте программу из первого примера так, чтобы в ней использовалась таблица Employees базы данных Northwind. Требуется извлечь столбцы EmployeeID и LastName.
2. Модифицируйте первую программу, демонстрирующую использование метода Update(), для того чтобы вернуть компании название Bottom-Dollar Markets.
3. Напишите программу, которая будет запрашивать у пользователя идентификатор клиента CustomerID.

4. Напишите программу для размещения нескольких заказов от имени клиента "Zachary Zithers"; используйте программу из примеров для просмотра этих заказов.

5. Напишите программу для вывода какой-либо другой части иерархии отношений в базе данных `Northwind`, например, между `Products`, `Suppliers` и `Categories`.

6. Запишите данные, полученные в результате выполнения предыдущего упражнения, в виде XML-документа.

7. Измените программу, применявшуюся для вывода всех составляющих заказа и наименований продукции для всех пользователей, использовав в операторе `SELECT` посылку `WHERE` для таблицы `Customers`, которое бы ограничивало количество обрабатываемых клиентов.

8. Модифицируйте программу, позволяющую выводить операторы `UPDATE`, `INSERT` и `DELETE`, для использования `"SELECT * FROM Customers"` в качестве SQL-команды `SELECT`. Обратите внимание на сложность сгенерированных выражений.



Работа с файлами

В этой главе вы узнаете, каким образом можно осуществлять считывание и запись текстовых файлов, что оказывается весьма существенным для многих приложений .NET. Мы будем обсуждать основные классы, которые используются для создания файлов, чтения из файлов и записи в них, а также вспомогательные классы, которые используются для выполнения различных манипуляций над файловой системой из программ на C#. Хотя у нас отсутствует возможность подробно описать все классы, наше рассмотрение окажется достаточно глубоким для того, чтобы вы получили хорошее представление об основных понятиях и принципах.

Файлы могут оказаться очень удобным средством сохранения информации при работе с несколькими вариантами приложения, они могут также использоваться для обмена данными между приложениями. Конфигурации пользователя и приложения также могут храниться в файлах, для того чтобы впоследствии извлекаться при очередном запуске приложения. Использование файлов может быть настолько разнообразным, насколько разными могут быть сами приложения. В .NET Framework предусмотрено наличие инструментов, позволяющих эффективно использовать файлы в приложениях.

В этой главе вы узнаете:

- Что такое поток, и каким образом в .NET используются классы для работы с потоками для доступа к файлам
- Как можно использовать объект `File` для выполнения манипуляций над структурой файла
- Как осуществляется запись в файл
- Как осуществляется чтение из файла
- Как осуществляется чтение из файлов и запись в файлы отформатированных данных
- Как можно осуществлять доступ к файлам синхронно (ко всем сразу) и асинхронно (по группам)
- Каким образом можно отслеживать изменения в файлах и директориях

Потоки

Любой ввод и вывод информации в .NET Framework включает в себя использование потоков — абстрактных представлений последовательного устройства. Последовательное устройство — это нечто, обеспечивающее как последовательное

хранение данных, так и последовательный доступ к ним — по одному байту в каждый конкретный момент времени. В качестве такого устройства могут выступать расположенный на диске файл, принтер, область памяти, а также любой другой объект, который допускает последовательное считывание и запись информации. Рассматривая такое устройство как абстрактное, мы можем скрыть назначение/источник потока. Такой уровень абстракции делает возможным многократное использование кода и позволяет создавать подпрограммы более общего назначения. Таким образом, аналогичный код может передаваться и повторно использоваться, когда приложение считывает данные из файлового входного потока и когда оно считывает информацию из сетевого входного потока. Кроме того, используя понятие потока, мы получаем возможность абстрагироваться от физических особенностей конкретного устройства. Поэтому при считывании информации из файлового потока нам не приходится волноваться по поводу работы головок дискового устройства или заниматься вопросами распределения памяти.

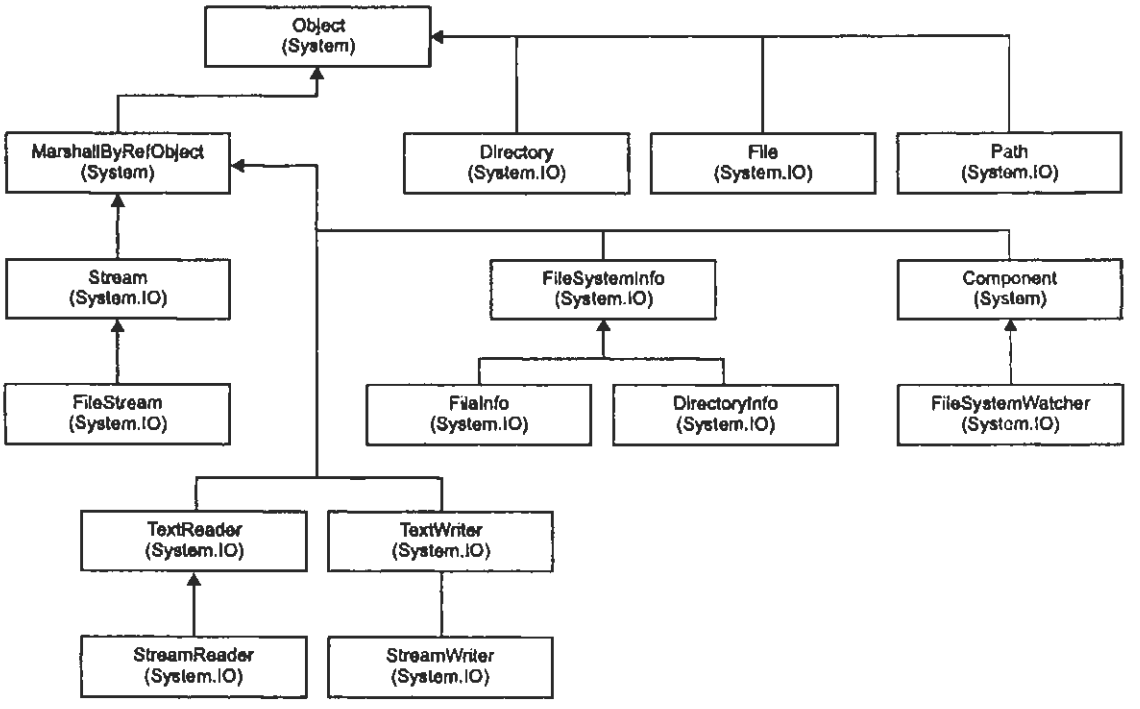
Выходной поток используется, когда данные записываются на некое внешнее устройство. Это может быть файл на физическом диске, сетевой сервер, принтер или какая-либо другая программа. Понимание программирования потоков открывает множество дополнительных возможностей. В этой главе мы ограничим обсуждение рамками записи в файлы на диске.

Входной поток используется для считывания данных в память или в переменные, к которым наша программа имеет доступ. Наиболее распространенной формой входного потока, с которой нам приходилось работать, является клавиатура. Входной поток может поступать практически от любого устройства, но мы сосредоточимся на чтении дисковых файлов. Понятия, применимые для чтения/записи дисковых файлов, оказываются применимыми практически для всех остальных устройств, что позволяет получить общее представление о потоках и познакомиться с фундаментальным подходом, который может быть использован в самых различных ситуациях.

Классы для ввода и вывода

Пространство имен `System.IO` содержит в себе все классы, которые будут рассматриваться в этой главе. В пространстве имен `System.IO` хранятся классы, предназначенные для считывания данных из файлов и записи данных в файлы; для того чтобы получить в программе доступ к этим классам, необходимо сослаться на это пространство имен. Как можно увидеть на следующей диаграмме, в пространстве имен содержится не так уж много классов, но мы будем рассматривать только классы, являющиеся основными при вводе и выводе файлов:

- `File` — широко используемый на практике класс, который обладает большим количеством статических методов, позволяющих переносить, копировать и удалять файлы.
- `Directory` — класс, который обладает большим количеством статических методов, позволяющих переносить, копировать и удалять директории.
- `Path` — класс, позволяющий выполнять манипуляции над именами путей.
- `FileInfo` — представляет физический файл, расположенный на диске, и обладает методами, позволяющими выполнять манипуляции над этим файлом. Для любых операций чтения/записи, выполняемых над файлом, необходимо создавать объект `Stream`.



- `DirectoryInfo` — представляет физическую директорию, расположенную на диске, и обладает методами, позволяющими выполнять манипуляции над этой директорией.
- `FileStream` — представляет файл, который допускает либо считывание, либо запись, либо и то, и другое одновременно. Этот файл может считываться и записываться как асинхронно, так и синхронно. Оба способа будут рассматриваться в настоящей главе.
- `StreamReader` — считывает символьную информацию из потока и может создаваться на базе класса `FileStream`.
- `StreamWriter` — записывает символьную информацию в поток и может создаваться на базе класса `FileStream`.
- `FileSystemWatcher` — это один из наиболее сложных классов, с которыми предстоит познакомиться в этой главе. Он используется для слежения за состоянием файлов и директорий и генерирует события в моменты, когда изменяется их местоположение, что позволяет перехватывать их внутри приложения. Такая функциональная возможность отсутствовала при программировании в среде Windows, но теперь, с появлением .NET Framework, реагировать на события, происходящие в файловой системе, становится намного проще.

Классы File и Directory

Оба класса предоставляют большое количество методов для осуществления манипуляций над самой файловой системой, а также над файлами и директориями, которые хранятся в этой файловой системе. Эти методы являются статическими и позволяют перемещать файлы, запрашивать и вносить изменения в атрибуты,

а также создавать объекты `FileStream`. Как мы узнали в главе 8, статические методы могут вызываться классами без создания их экземпляров.

Некоторые из наиболее полезных статических методов класса `File` приведены в таблице справа.

В следующей ниже таблице представлены некоторые полезные статические методы класса `Directory`:

Метод	Описание
`Copy()`	Копирует файл в указанное место.
`Create()`	Создает файл в соответствии с заданным путем.
`Delete()`	Уничтожает файл.
`Open()`	Возвращает объект `FileStream` по заданному пути.
`Move()`	Перемещает файл в новое место. При этом имеется возможность задать для нового файла другое имя.

Метод	Описание
`CreateDirectory()`	Создает директорию в соответствии с заданным путем.
`Delete()`	Уничтожает указанную директорию вместе со всеми находящимися в ней файлами.
`GetDirectories()`	Возвращает массив объектов `Directory`, в котором представлены все директории, находящиеся в данной директории.
`GetFiles()`	Возвращает массив объектов `File` для данной директории.
`Move()`	Переносит указанную директорию на новое место, где этой папке может быть присвоено новое имя.

Класс FileInfo

В отличие от класса `File`, класс `FileInfo` не обладает статическими методами и может использоваться только посредством создания экземпляров объектов. Объект `FileInfo` представляет файл, расположенный на диске или по некоторому сетевому адресу. Заметьте, что этот класс не является потоком. Для того чтобы осуществить запись в файл или чтение из файла, необходимо создать объект `Stream`. Объект `FileInfo` помогает решить эту задачу, предоставляя несколько методов, которые возвращают созданные экземпляры объектов `Stream`. Однако прежде всего, для того чтобы создать объект `FileInfo`, необходимо указать путь к файлу или директории:

```
FileInfo aFile = new FileInfo("C:/log.txt");
```

Многие методы, предоставляемые классом `FileInfo`, аналогичны соответствующим методам класса `File`, однако, поскольку класс `File` является статическим классом, он требует задания строкового параметра с местоположением файла при каждом вызове метода. Отсюда следует, что следующие вызовы выполняют одни и те же действия:

```
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 /
```

Большинство методов объекта `FileInfo` представляют в этом смысле зеркальное отражение методов объекта `File`. Имеет прямой смысл использовать статический класс `File`, когда требуется осуществить единственный вызов метода на объект. В этом случае вызов будет выполнен быстрее, поскольку .NET Framework не придется проходить через процедуру создания экземпляра нового объекта с последующим вызовом метода. Однако если приложение осуществляет несколько операций над файлом, то более разумным представляется создать экземпляр объекта `FileInfo` и использовать его методы. Это позволит сэкономить определенное

Свойство	Описание
Attributes	Позволяет получать или задавать значения атрибутов текущего файла.
CreationTime	Позволяет получать дату и время создания текущего файла.
DirectoryName	Возвращает путь к директории, в которой находится файл.
Exists	Определяет, существует ли данный файл.
FullName	Позволяет получать полный путь к данному файлу.
Length	Позволяет получать размер файла.
Name	Возвращает только имя самого файла — без полного пути, ведущего к его местоположению.

время, поскольку объект будет заранее настроен на нужный файл в файловой системе, в то время как статическому классу придется каждый раз осуществлять его поиск заново.

Класс `FileInfo` обладает перечисленными в таблице слева свойствами, относящимися к данному файлу, осуществление манипуляций над которыми позволяет вносить изменения в файл.

Класс DirectoryInfo

Класс `DirectoryInfo` работает в точности так же, как и класс `FileInfo`. Он требует создания экземпляра объекта, который представляет собой одну из директорий, имеющихся в компьютере. Как и для класса `FileInfo`, многие методы дублируются в классах `Directory` и `DirectoryInfo`. Аналогичны и указания относительно того, когда какой из них следует использовать: при единственном вызове следует использовать статический класс `Directory`. При осуществлении последовательности вызовов нужно использовать экземпляр объекта `DirectoryInfo`.

Имена пути и относительные пути

При задании имени пути в программе для .NET имеется возможность использовать либо абсолютное, либо относительное имя пути. Абсолютное имя пути явным образом указывает на расположение файла или директории в каком-либо известном месте, например, на диске C:. Вот пример такого пути: `C:\Work\LogFile.txt`. Обратите внимание на то, что в данном случае местоположение описывается точно, без какой-либо двусмысленности.

Относительные пути указывают на местоположение относительно того места в файловой системе, в котором выполняется приложение. При использовании относительного пути не требуется указывать ни имени диска, ни какого-либо известного места; в качестве отправной точки используется текущая директория. Например, если приложение выполняется в директории `C:\Development\FileDemo`, а в качестве относительного пути используется `"LogFile.txt"`, то это будет файл `C:\Development\FileDemo\LogFile.txt`. Для перехода к директории более высокого уровня используется две точки (..). Следовательно, для того же самого приложения путь `"..\Log.txt"` будет указывать на файл с именем `Log.txt`, расположенный в директории `C:\Development`.

Наиболее коварным при использовании относительных путей в процессе разработки является именно их относительность, зависящая от того, где именно выполняется данное приложение. Если разработка ведется с использованием Visual Studio.NET, то это означает, что приложение находится на несколько уровней ниже создаваемой вами папки проекта. Обычно приложения располагаются в директории `ProjectName\bin\Debug`.

Объект FileStream

Объект `FileStream` представляет поток, указывающий на какой-либо файл на диске или на местоположение в сети. Хотя этот класс и обладает методами для чтения из файлов и записи в файлы байтов, чаще для выполнения этих функций

вам придется обращаться к классам `StreamReader` и `StreamWriter`. Это происходит потому, что класс `FileStream` оперирует байтами и массивами байтов, в то время как классы `Stream` оперируют символьными данными. Иметь дело с символьными данными оказывается проще, однако нам придется столкнуться с некоторыми операциями, например, с произвольным доступом к файлам, которые могут осуществляться только посредством объекта `FileStream`, к изучению которого мы приступим несколько позже.

Существует несколько способов создания объекта `FileStream`. Его конструктор обладает большим количеством перегрузок/версий, однако в самом простом случае он требует передачи двух аргументов — имени файла и значения перечислимого типа `FileMode`:

```
FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate);
```

У перечислимого типа `FileMode` имеется несколько членов, которые позволяют задавать то, каким образом файл открывается или создается. Допускается их использование в различных сочетаниях.

Элементы перечислимого типа FileMode	Описание
`Append`	Открывает файл (если он существует), переводит указатель файла в его конец или создает новый файл. `FileMode.Append` может использоваться только совместно с членом перечислимого типа `FileAccess.Write`.
`Create`	Создает новый файл; если файл с таким именем уже существует, то он уничтожается.
`CreateNew`	Создает новый файл; если файл с таким именем уже существует, то будет сгенерирована исключительная ситуация.
`Open`	Открывает существующий файл. Если указанного файла не существует, генерируется исключительная ситуация.
`OpenOrCreate`	Определяет, что если такой файл существует, то он должен быть открыт, в противном случае создается новый файл. Если такой файл существует, вся информация в нем сохраняется.
`Truncate`	Открывает существующий файл, а вся информация, которая в нем хранилась, уничтожается. После этого мы получаем возможность записать в файл абсолютно новую информацию, однако при этом дата создания файла остается прежней. Этот файл должен существовать, иначе будет сгенерирована исключительная ситуация.

Приведенный выше пример конструктора по умолчанию открывает файл в режиме чтение/запись. Для того чтобы задать другой уровень доступа, следует использовать дополнительный параметр — `FileAccess`:

```
FileStream aFile = new FileStream("Log.txt", FileMode.OpenOrCreate,
                                  FileAccess.Write);
```

Эта строка кода откроет файл с разрешением доступа на запись. Любая попытка прочитать содержимое этого файла приведет к возникновению исключительной ситуации. Перечислимый тип `FileAccess` обладает всего тремя типами: `Read`, `Write` и `ReadWrite`. Следовательно, есть возможность открыть файл только на чтение, только на запись и на чтение и запись одновременно. Это свойство используется в качестве одного из способов варьирования уровня доступа пользователя к файлу в зависимости от уровня его прав.

Оба класса `File` и `FileInfo` предоставляют методы `OpenRead()` и `OpenWrite()`, которые облегчают создание объектов `FileStream`. Первый открывает метод с режимом доступа "только чтение", а второй, кроме того, разрешает осуществлять запись. Это позволяет упростить обращение к файлу, так что вам теперь не придется указывать всю информацию, как в предыдущих примерах. Например, следующая строка открывает файл `Data.txt` в режиме "только чтение":

```
FileStream aFile = File.OpenRead("Data.txt");
```

Обратите внимание на то, что следующий код выполняет те же самые действия:

```
FileInfo aFile = new FileInfo("Data.txt");
FileStream aFile = aFile OpenRead();
```

Позиция внутри файла

Класс `FileStream` поддерживает внутренний указатель файла, ссылающийся на то место в файле, в котором будет производиться очередная операция чтения или записи. В большинстве случаев при открытии файла указатель устанавливается в начало файла, однако такое поведение указателя может быть изменено, что позволяет приложению осуществлять чтение или запись в любой точке файла. Можно также организовать произвольный доступ к файлу и осуществлять поиск конкретной позиции в файле. Такой подход позволяет сэкономить массу времени при работе с очень большими файлами, поскольку можно постоянно отыскивать необходимую позицию.

Метод, который реализует данную функциональную возможность, носит название `Seek()`:

```
public long Seek(long offset, SeekOrigin origin);
```

Первый параметр определяет, насколько далеко вперед в байтах должен быть передвинут указатель. Второй параметр определяет, с какой точки вести отсчет. Перечислимый тип `SeekOrigin` состоит из трех значений: `Begin` (начало), `Current` (текущая позиция) и `End` (конец). Следующая строка передвигает указатель файла на восьмой байт, считая с самого первого байта данного файла:

```
aFile.Seek(8, SeekOrigin.Begin);
```

Следующая строка кода приведет к перемещению указателя файла на два байта вперед относительно его текущей позиции. Если эта строка будет выполнена непосредственно сразу после предыдущей строки, то указатель будет ссылаться на десятый байт файла:

```
aFile.Seek(2, SeekOrigin.Current);
```

Следует обратить внимание на то, что как при чтении из файла, так и при записи в файл происходит изменение позиции указателя файла. В случае считывания, например, десяти байтов, указатель файла будет ссылаться на байт, расположенный после десятого считанного байта.

Существует также возможность указывать отрицательные значения для определения требуемой позиции, что в сочетании с использованием значения перечислимого типа `SeekOrigin.End` может быть использовано для поиска необходимой позиции вблизи конца файла. Следующая строка позволит перейти к пятому с конца байту файла:

```
aFile.Seek(-5, SeekOrigin.End);
```

Файлы, доступ к которым может осуществляться подобным образом, зачастую называют файлами с произвольным доступом, поскольку у прнложения имеется возможность получить доступ к любой позиции в файле. Классы `Stream` позволяют осуществлять последовательный доступ к файлам, и в них не предусмотрена возможность работы с указателем.

Чтение данных

Чтение данных с использоваинем класса `FileStream` оказывается не таким простым, как с использованием классов `StreamReader` и `StreamWriter`, к рассмотрению которых мы вернемся ннже в данной главе. Это происходит потому, что класс `FileStream` имеет дело исключительно с байтами, что делает класс полезным при работе с файлами произвольного типа, а не только с текстовыми файлами. Побайтное счнтывание данных позволяет использовать этот класс при работе с графическнми и звуковыми файлами. Однако платой за подобную гибкость является невозможность с помощью класса `FileStream` считывать данные непосредственно в переменные типа строки. Это позволяют осуществлять классы `StreamWriter` и `StreamReader`. Существует однако несколько конверсионных классов, которые существенно облегчают выполнение преобразования массива байтов в символьные массивы и обратно.

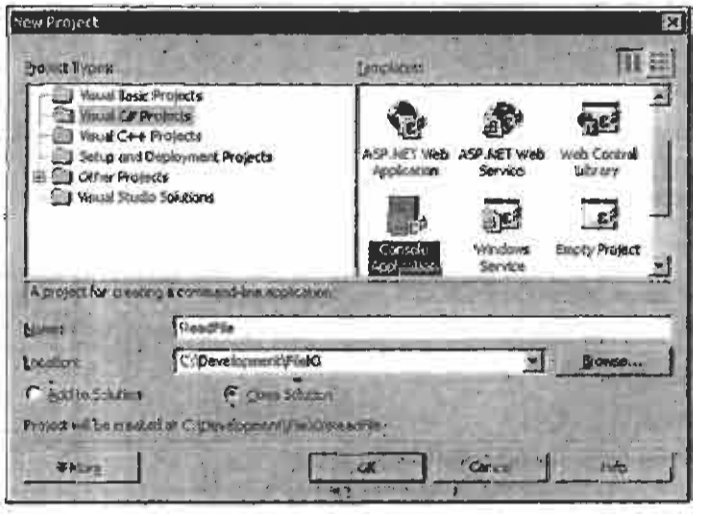
Метод `FileStream.Read()` является основным средством, позволяющим осуществлять доступ к данным, содержащнмся в файле, на который ссылается объект `FileStream`:

```
public int Read(byte[] array, int offset, int count);
```

Первый параметр представляет собой массив байтов, передаваемый для размещения в нем данных, получаемых от объекта `FileStream`. Второй параметр определяет ту позицню в массиве байтов, начиная с которой должна осуществляться операция записи данных. Обычно этот параметр имеет значение, равное нулю, в результате чего данные нз файла заносятся в самое начало массива. Последний параметр определяет, какое количество байтов должно быть считано из файла.

Следующий пример демоистрирует считывание даниых из файла с произвольным доступом. В качестве файла, из которого будет производиться чтение, используется файл, специально созданный для данного примера.

Практикум: чтение данных из файла с произвольным доступом



1. Откройте Visual Studio.NET.
2. Откройте новый проект, выбрав **File | New | Project**. Выберите пункт **C# Console Application** и назовите его **ReadFile** (см. рис. слева).
3. Добавьте два следующих объявления пространств имен в самое начало файла `Class1.cs`. Пространство имеи `System.IO` необходимо для класса `FileStream`, а пространство имен `System.Text` потребуется для осуществления

преобразования, которое нам придется выполнять над байтами, считанными из файла.

```
using System;
using System.IO;
using System.Text;
```

4. Добавьте следующий код в метод Main():

```
static void Main(string[] args)
{
   byte[] byData = new byte[100];
   char[] charData = new Char[100];
   try
   {
      FileStream aFile = new FileStream("../../Class1.cs",FileMode.Open);
      aFile.Seek(55,SeekOrigin.Begin);
      aFile.Read(byData,0,100);
   }
   catch(IOException e)
   {
      Console.WriteLine("An IO exception has been thrown!");
      Console.WritsLine(e.ToString());
      Console.ReadLine();
      return;
   }

   Decoder d = Encoding.UTF8.GetDecoder();
   d.GetChars(byData, 0, byData.Length, charData, 0);

   Console.WriteLine(charData);
   Console.ReadLine();

   return;
}
```



5. Запустите приложение. Вы должны увидеть приблизительно такой выходной поток, какой показан на рисунке слева. Имейте в виду, что выходной поток на вашем компьютере может существенно отличаться от изображенного на рисунке. Нажмите клавишу *Enter* для закрытия консоли.

Как это работает — Чтение данных из файла с произвольным доступом

Данное приложение открывает свой собственный файл с расширением .cs, из которого затем осуществляется чтение. Для этого оно осуществляет поиск по файловой структуре с использованием двух точек в следующей строке:

```
FileStream aFile = new FileStream("../../Class1.cs", FileMode.Open);
```

Две строки кода, которые в действительности осуществляют поиск по файлу и считывание из него, имеют следующий вид:

```
try
{
   aFile.Seek(55,SeekOrigin.Begin);
   aFile.Read(byData,0,100);
}
```

```
catch(IOException e)
{
   Console.WriteLine("An IO exception has been thrown!");
   Console.WriteLine(e.ToString());
   Console.ReadLine();
   return;
}
```

В результате выполнения первой из них указатель файла перемещается на 55-й байт файла. Это символ 'n' из слова `namespace`; 54 предыдущих символа находятся в строках с директивами `using`. В результате выполнения второй строки следующие 100 байтов считываются в байтовый массив `byData`.

Обратите внимание на то, что нами используется конструкция для обработки исключительных ситуаций `try-catch`, с которой мы познакомились в главе 7. Практически любая операция, связанная с вводом/выводом, может привести к возникновению исключительной ситуации типа `IOException`. Любой исполнимый код должен включать в себя обработку исключительных ситуаций, особенно если в нем ведется работа с файловой системой. Во всех примерах настоящей главы будет использоваться некоторый основополагающий способ обработки ошибок.

После того как информация считана из файла в байтовый массив, возникает необходимость преобразовать ее в символьный массив с тем, чтобы получить возможность выводить ее на консоль. С этой целью мы воспользуемся классом `Decoder` из пространства имен `System.Text`. Этот класс предназначен для того, чтобы преобразовывать исходные байты в какие-либо более полезные объекты, например, в символы:

```
Decoder d = Encoding.UTF8.GetDecoder();
d.GetChars(byData, 0, byData.Length, charData, 0);
```

В этих строках осуществляется создание объекта `Decoder`, который основывается на кодировке UTF8. Это схема кодирования Unicode. После этого вызывается метод `GetChars()`, который принимает массив байтов и преобразовывает его в массив символов. После завершения этой процедуры массив символов оказывается готовым к выводу на консоль.

Запись данных

Процесс записи данных в файл с произвольным доступом совершенно аналогичен. Сначала необходимо создать байтовый массив; наиболее простой способ достигнуть этого — создать сначала символьный массив, который мы собираемся записывать в файл. Затем следует воспользоваться объектом `Encoder`, позволяющим преобразовать его в массив байтов; использование этого объекта очень напоминает использование объекта `Decoder`. В завершение необходимо вызывать метод `Write()`, который и запишет информацию в файл.

Давайте создадим простой пример, который позволит продемонстрировать, каким образом это выполняется.

Практикум: запись данных в файл с произвольным доступом

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его `WriteFile`.

2. Как и в предыдущем случае, вставьте в начало файла `Class1.cs` следующие два описания пространств имен:

```
using System;
using System.IO;
using System.Text;
```

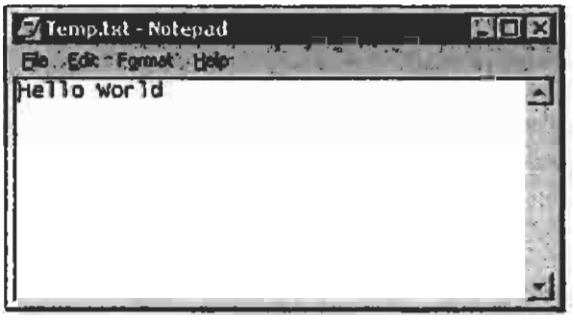
3. Добавьте в метод Main() следующий код:

```
static void Main(string[] args)
{
   byte[] byData = new byte[100];
   char[] charData = new Char[100];

   try
   {
      FileStream aFile = new FileStream("Temp.txt",FileMode.OpenOrCreate);
      charData = "Hello World".ToCharArray();
      Encoder e = Encoding.UTF8.GetEncoder();
      e.GetBytes(charData,0,charData.Length, byData,0,true);

      // Перемещение указателя файла в самое начало файла
      aFile.Seek(0,SeekOrigin.Begin);
      aFile.Write(byData,0,byData.Length);
   }
   catch(IOException ex)
   {
      Console.WriteLine("An IO exception has been thrown!");
      Console.WriteLine(ex.ToString());
      Console.ReadLine();
      return;
   }

   return;
}
```



4. Запустите приложение. Оно будет выполняться некоторое время, а затем завершится.

5. Перейдите в директорию приложения — файл будет сохранен именно в ней, поскольку мы используем относительный путь. Файл будет помещен в папку WriteFile\bin\Debug. Откройте файл Temp.txt. Вы сможете обнаружить в файле текст, приведенный на рисунке слева.

Как это работает — Запись данных в файл с произвольным доступом

Данное приложение открывает файл в своей собственной директории и записывает в него несложную строку. Все это происходит аналогично предшествующему примеру.

Следующая строка создает символьный массив с помощью статического метода ToCharArray() класса String. Поскольку в C# все можно рассматривать в качестве объекта, а текст "Hello World" на самом деле представляет собой объект String, то эти методы могут вызываться даже для обычной строки символов:

```
charData = "Hello World".ToCharArray();
```

Из следующих строк становится понятным, каким образом можно преобразовать массив символов в соответствующий массив байтов, который требуется объекту FileStream.

```
Encoder e = Encoding.UTF8.GetEncoder();
e.GetBytes(charData,0,charData.Length, byDate,0,true;
```

На этот раз мы создаем объект `Encoder`, который также основывается на кодировке UTF8. В данном случае для выполнения перекодировки также используется кодировка Unicode, однако на этот раз нам необходимо перекодировать символьные данные в корректный байтовый формат, прежде чем у нас появится возможность записать информацию в поток. Это совершается в методе `GetBytes()`. В нем происходит преобразование символьного массива в байтовый. Этому методу в качестве первого параметра передается символьный массив, а в качестве второго — индекс элемента, с которого следует начинать преобразование. В качестве третьего параметра этому методу передается количество символов, подлежащих преобразованию; в качестве четвертого — байтовый массив, в котором должна размещаться информация после преобразования, и в качестве пятого — индекс элемента, с которого начинается заполнение байтового массива. Последний параметр определяет, должен ли объект `Encoder` обновлять свое состояние после окончания работы. Это имеет отношение к тому факту, что объект `Encoder` сохраняет в памяти то место, на котором была прекращена запись в байтовый массив. Это оказывается весьма полезным при осуществлении последовательных обращений к объекту `Encoder`, но оказывается совершенно бессмысленным в случае единственного обращения к нему. При завершающем обращении к объекту `Encoder` этому параметру должно быть присвоено значение `true`, означающее, что память должна быть очищена, а объект освобожден для процедуры сборки мусора.

После выполнения всех этих действий запись байтового массива в объект `FileStream` посредством метода `Write()` не представляет никаких трудностей:

```
aFile.Seek(0,SeekOrigin.Begin);
aFile.Write(byData,0,byData.Length);
```

Объект StreamWriter

Работа с объектом `FileStream` достаточна сложна, поэтому может возникнуть вопрос, нет ли какого-либо более простого способа. Не стоит беспокоиться — обычно вся работа с объектом `FileStream` оказывается скрытой от пользователя за счет работы с объектами `StreamWriter` и `StreamReader` и использования их методов для выполнения манипуляций над файлами. Если перед вами не стоит задача перемещать указатель файла на произвольную позицию, то здесь эти классы позволяют значительно упростить работу с файлами. Класс `StreamWriter` позволяет осуществлять запись в файл символов и строк, он самостоятельно выполняет все необходимые преобразования и производит запись в объект `FileStream`.

Существуют разнообразные способы создания экземпляров класса `StreamWriter`. Если уже имеется объект `FileStream`, то можно воспользоваться им для создания объекта `StreamWriter`:

```
FileStream aFile = new FileStream("Log.txt",FileMode.CreateNew);
StreamWriter sw = new StreamWriter(aFile);
```

Объект `StreamWriter` может также создаваться непосредственно из файла:

```
StreamWriter sw = new StreamWriter("Log.txt",true);
```

Этот конструктор принимает в качестве параметров имя файла и логическое значение, которое указывает на то, следует ли добавлять информацию к уже существующему файлу или необходимо создать новый. Если это значение равно `false`, то либо происходит создание нового файла, либо из существующего файла сначала удаляется все хранящаяся в нем информация, а затем он открывается на запись. Если же это значение равно `true`, то файл просто открывается, а все хранящаяся

в нем информация сохраняется. Если такого файла не существует, то создается новый файл.

Обратите внимание на то, что здесь нет такого широкого спектра возможностей, как при создании объекта `FileStream`. Кроме логического значения, определяющего, следует ли добавлять информацию к уже существующему файлу или создавать новый файл, нет никаких других опций, например, отсутствует возможность задания свойства `FileMode`, которое мы уже задавали при работе с классом `FileStream`. Точно так же отсутствует возможность задания свойства `FileAccess`, поэтому вы всегда будете обладать доступом к файлу на чтение/запись. Для того чтобы воспользоваться какой-либо из этих возможностей, следует сначала обратиться к конструктору класса `FileStream`, а затем создать объект `StreamWriter` на основе объекта `FileStream`.

Практикум: выходной поток

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его `StreamWrite`.

2. В данном случае опять придется воспользоваться пространством имен `System.IO`, поэтому добавьте соответствующую строку в начало файла `Classl.cs`.

```
using System;
using System.IO;
```

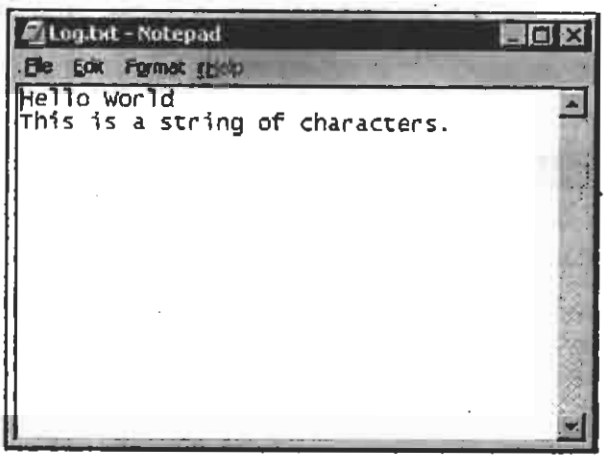
3. Добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
    try
    {
        FileStream aFile = new FileStream("Log.txt",FileMode.OpenOrCreate);
        StreamWriter sw = new StreamWriter(aFile);

        // Запись данных в файл
        sw.WriteLine("Hello World");
        sw.Write("This is a ");
        sw.Write("string of characters.");
        sw.Close();
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }
    return;
}
```

4. Постройте проект и запустите его. Если в нем не будет обнаружено никаких ошибок, то он очень быстро выполнится и закроется. Поскольку мы не предусмотрели вывода на консоль, то наблюдать за работой этой программы оказывается не очень интересно.

5. Перейдите в директорию приложения и найдите там файл `Log.txt`. Он должен располагаться в папке `StreamWriter\bin\Debug`, поскольку используется относительный путь.



6. После того как вы откроете этот файл, вы увидите символы, приведенные на рисунке справа.

Как это работает — Выходной поток

Данное простое приложение демонстрирует работу двух наиболее важных методов класса `StreamWriter` — `Write()` и `WriteLine()`. У каждого из них имеется большое количество перегружаемых версий, позволяющих использовать различные сложные возможности при выводе в файл, однако в настоящем примере мы воспользовались основной возможностью вывода строки.

Метод `WriteLine()` осуществляет запись переданной строки, за которой немедленно следует символ новой строки. Как мы можем видеть в настоящем примере, это приводит к тому, что следующая операция записи выполняется с новой строки.

Метод `Write()` просто записывает передаваемую ему строку в файл, не добавляя символа перехода к новой строке. Это позволяет осуществлять запись целого предложения или абзаца с помощью нескольких обращений к методу `Write()`.

Форматирование данных

Точно так же, как существует возможность вывода данных в отформатированном виде на консоль, так и в файл мы можем записывать отформатированные данные. Для этого применяются те же методы — `Write()` и `WriteLine()`, но в этом случае им могут передаваться параметры, определяющие характер форматирования. Например:

```
sw.WriteLine("The Date is {0}", System.DateTime.Now.ToShortDateString());
sw.Write("{0}, {1}, {2}", "Kay", "Dan", "Rob");
```

В данном примере вы можете увидеть, каким образом осуществляется запись файлов, разделенных запятыми. В более сложных примерах данные могут поступать из объекта `DataSet` и других источников.

Объект StreamReader

Входные потоки используются для считывания данных, хранящихся во внешних источниках. В большинстве случаев таким источником является дисковый файл или сетевой адрес. Однако не следует забывать, что в качестве такого источника может использоваться любой объект, отправляющий данные, например, сетевое приложение, web-служба и даже консоль.

Для чтения данных мы будем использовать класс `StreamReader`. Так же, как и класс `StreamWriter`, класс `StreamReader` является общим классом, который может использоваться для любого потока. Мы снова будем создавать его на базе объекта `FileStream`, поэтому он будет ссылаться на нужный нам файл.

Объекты `StreamReader` создаются способом, напоминающим способ создания объектов `StreamWriter`. Наиболее распространенный способ создания таких объектов заключается в использовании заранее созданного объекта `FileStream`:

```
FileStream aFile = new FileStream("Log.txt",FileMode.Open);
StreamReader sr = new StreamReader(aFile);
```

Так же, как и класс `StreamWriter`, `StreamReader` может создаваться непосредственно на основе строки, в которой содержится путь к конкретному файлу:

```
StreamReader sr = new StreamReader("Log.txt");
```

Практикум: входной поток

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его `StreamRead`.

2. В данном случае нам опять потребуется импортировать пространство имен `System.IO`, поэтому добавьте соответствующую строку в начало файла `Class1.cs`.

```
using System;
using System.IO;
```

3. Добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
    string strLine;
    try
    {
        FileStream aFile = new FileStream("Log.txt",FileMode.Open);
        StreamReader sr = new StreamReader(aFile);
        strLine = sr.ReadLine();
        //Построчное считывание данных
        while(strLine != null)
        {
            Console.WriteLine(strLine);
            strLine = sr.ReadLine();
        }
        sr.Close();
        Console.ReadLine();
    }
    catch(IOException e)
    {
        Console.WriteLine("An IO exception has been thrown!");
        Console.WriteLine(e.ToString());
        Console.ReadLine();
        return;
    }
    return;
}
```

4. Создайте в директории `StreamReader\bin\Debug` файл `Log.txt`. Это может быть совершенно новый файл или скопированный регистрационный файл, созданный в предыдущем примере. Убедитесь, что файл находится в той же директории, что и приложение! В противном случае приложение завершится неудачей, поскольку конструктор класса `FileStream` сгенерирует исключительную ситуацию, если ему не удастся обнаружить требуемый файл.



5. Запустите приложение — текст, хранящийся в файле, будет выведен на экран, как показано на рисунке слева.

Как это работает — Входной поток

Данное приложение очень похоже на предыдущее. Единственное очевидное отличие заключается в том, что в нем происходит считывание информации из файла, а не запись. В этом случае также необходимо импортировать пространство имен `System.IO`, для того чтобы осуществлять доступ к необходимым классам.

Для чтения текста из файла использован метод `ReadLine()`. Этот метод осуществляет считывание текста до того момента, когда будет обнаружен символ "возврат каретки", и возвращает считанный текст в виде строки. В случае, если будет достигнут конец файла, то данный метод возвращает значение `null`, что используется при выполнении проверки достижения конца файла:

```
strLine = sr.ReadLine();
while(strLine != null)
{
   Console.WriteLine(strLine);
   strLine = sr.ReadLine();
}
```

Здесь используется метод `Console.LeadLimene()`, чтобы консоль не завершила свою работу, до того как мы успели прочитать выведенную на дисплей итоговую информацию.

Чтение данных

Метод `ReadLine()` не является единственным способом доступа к данным в файле. Класс `StreamReader` имеет несколько способов для чтения данных.

Простейший из них — `Read()`. Он возвращает следующий символ в виде положительного целого числа или значения –1, если достигнут конец. С помощью переводного класса `Convert` это значение может быть преобразовано в символ. В приведенном выше примере основная часть программы может быть переписана следующим образом:

```
int nChar;
nChar = sr.Read();
while(nChar != -1)
{
   Console.Write(Convert.ToChar(nChar));
   nChar = sr.Read();
}
```

Удобным методом для чтения небольших файлов является метод `ReadToEnd()`, осуществляющий считывание всего файла целиком и возвращающий его в виде строки. В этом случае приведенное выше приложение упрощается следующим образом:

```
strLine = sr.ReadToEnd();
Console.WriteLine(strLine);
```

Хотя такой подход может показаться весьма простым и удобным, при использовании его необходимо соблюдать известную осторожность. Производя считывание всех данных в строковый объект, вы форсируете перевод данных, хранящихся в файле, в память компьютера. В зависимости от размера файла с данными, это может оказаться невыполнимым. Если файл данных достаточно велик, то надежнее будет оставить данные в файле и осуществлять доступ к ним с помощью методов объекта `StreamReader`.

Файлы с ограничителем

Распространенным способом хранения данных в файле является разделение данных запятыми. Мы уже сталкивались с использованием класса `StreamWriter` для записи файлов с использованием такого подхода. Не менее просто оказывается читать файлы, использующие запятые в качестве разделителя. В класс `String` включен метод с названием `Split()`, который используется для преобразования строки в массив с использованием заданного символа в качестве разделителя. Если мы зададим режим использования запятой в качестве разделителя, то это приведет к созданию строкового массива требуемой размерности, в котором будет содержаться вся информация из исходной строки.

Практикум: файлы с использованием запятой в качестве разделителя

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его `CommaValues`.

2. Добавьте соответствующую строку в начало файла `Class1.cs`:

```
using System;
using System.IO;
```

3. Добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
   string strLine;
   string[] strArray;
   char[] charArray = new Char[] (',');

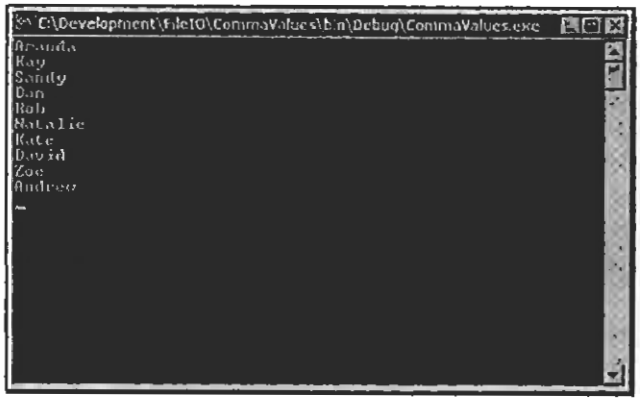
   try
   {
      FileStream aFile = new FileStream("Log.txt",FileMode.Open);
      StreamReader sr = new StreamReader(aFile);

      strLine = sr.ReadLine();
      while(strLine != null)
      {
         // Преобразование строк данных в массив строк
         strArray = strLine.Split(charArray);

         // Проход в цикле по массиву строк и вывод их на печать
         for(int x=0;x<=strArray.GetUpperBound(0);x++)
         {
            Console.WriteLine(strArray[x].Trim());
         }
         strLine = sr.ReadLine();
      }

      sr.Close();
      Console.ReadLine();
   }
   catch(IOException e)
   {
      Console.WriteLine("An IO exception has been thrown!");
      Console.WriteLine(e.ToString());
      Console.ReadLine();
      return;
   }

   return;
}
```



4. Откройте Notepad и введите в него следующий текст:

   Amanda, Kay, Sandy, Dan, Rob
   Natalie, Kate, David, Zoe, Andrew

5. Сохраните файл под именем `log.txt` в директории `CommaValues\bin\Debug` или директории, где будет запускаться выполняемый файл вашего приложения.

6. Запустите приложенне — вы увидите выведенный на консоль текст файла (см. рис. справа).

Как это работает — Файлы с использованием запятой в качестве разделителя

Как и в предыдущем примере, данное приложение осуществляет построчное считывание данных в объект `String`. Однако, поскольку заранее известно, что в этом файле в качестве разделителей текстовых значений применяются запятые, то нами используется другой подход для его обработки. В следующей строке программы происходит создание строкового массива для каждой строки файла:

```
strArray = strLine.Split(charArray);
```

Данный метод принимает строковый массив, созданный ранее. Обратите внимание, что в настоящем примере в массиве содержится единственный снмвол — запятая. Мы могли бы записать в массив несколько символов; в таком случае каждый раз, когда в файле будет встречен любой из указанных символов, он будет рассматриваться в качестве допустимого ограничителя поля. Например, в следующем коде создается символьный массив, в который помещается сразу несколько символов. Теперь, в случае если метод `Split` будет вызван для строки `"Please split?this|String up"`, то в строковом массиве будут возвращены все слова по отдельности, поскольку все символы, расположенные между словами, будут рассматриваться методом `Split` в качестве разделителей (пробел, вопросительный знак и вертикальная черта):

```
char[] charArray = new Char[] {'?',' ','|'};
strArray = "Please split?this|String up".Split(charArray);
```

После получения строкового массива он просматривается в цикле, а его содержимое выводится на консоль. Это достигается за счет использования обычного цикла `for`. Обратите внимание на то, что мы вызываем метод `Trim()` класса `String`. Этот метод удаляет все пустое пространство (пробелы и символы табуляции) в начале и в конце строк. Мы делаем это, поскольку нас не интересуют пробелы в начале и конце данных — нас интересуют только сами имена. Такой подход не всегда подойдет для ваших приложений, поскольку пустое пространство может оказаться значимым:

```
for(int x=0;x<=strArray.GetUpperBound(0);x++)
{
   Console.WriteLine(strArray[x].Trim());
}
```

Как легко заметить, извлечение данных из файлов, в которых в качестве разделителя переменных используется запятая (comma-separated variable — CSV),

в .NET Framework не представляет никакого труда. Хотя в данном случае приведенный пример оказывается очень простым, вы столкнетесь с тем, что эти принципы используются в бизнес-приложениях уровня целого предприятия. Несмотря на то, что XML является намного превосходящим по всем параметрам методом хранения и передачи данных, использование CSV-файлов по-прежнему очень распространено и будет распространено еще в течение некоторого времени. Преимуществом файлов, в которых используются ограничители, заключается еще и в том, что они являются очень компактными и, следовательно, занимают меньше места, чем их XML-аналоги.

Файлы с ограничителями используются многими системами, и если вашему приложению придется взаимодействовать с такой системой, то вы обнаружите, что формат данных с использованием ограничителей применяется достаточно часто. Обратите внимание на то, что, хотя мы и уделяли основное внимание файлам, в которых в качестве разделителя используется запятая, в этом качестве вполне может использоваться и символ табуляции — такой формат также является весьма распространенным. Следует также помнить, что CSV-файлы могут экспортироваться и импортироваться такими системами, как Excel, Access и многими реляционными базами данных.

Дополнительные темы

В этом разделе мы рассмотрим вопрос, касающийся наблюдения за состоянием файловой системы, в результате которого приложение сможет получать уведомление об происходящих изменениях, а также асинхронный доступ к файлам, который позволяет загружать большие файлы на фоне продолжения выполнения приложением других процессов.

Наблюдение за состоянием файловой структуры

Иногда в задачу приложения входит нечто большее, чем просто чтение и запись файлов, хранящихся в файловой системе, и оказывается важным определить момент, когда в структуру файлов и директорий вносятся изменения. .NET Framework существенно облегчает создание приложений, которые могут выполнять поставленную задачу.

Класс, позволяющий нам добиваться этого, называется `FileSystemWatcher`. Он обладает несколькими событиями, которые могут перехватываться приложениями, что дает им возможность реагировать на такие системные события соответствующим образом.

Основной способ использования класса `FileSystemWatcher` весьма прост. Сначала нужно задать значения свойств, с помощью которых можно определить, где следует осуществлять контроль, что при этом необходимо контролировать и когда следует генерировать события, которые будут обрабатываться нашим приложением. Затем сообщить адреса созданных обработчиков событий, с тем чтобы они могли вызываться в момент наступления значимых событий. Наконец, следует запустить программу и перейти в режим ожидания наступления соответствующих событий.

В таблице на стр. 575 приведены свойства, которые необходимо задать до того, как появится возможность использовать объект `FileSystemWatcher`.

После того как эти свойства будут заданы, необходимо создать обработчики событий для четырех различных событий — `Changed`, `Created`, `Deleted` и `Renamed`. Как известно из главы 12, для этого необходимо просто создать свой собственный

метод и присвоить его событию данного объекта. Присваивание вашего собственного обработчика событий этим методам приведет к тому, что именно ваш метод будет вызываться при наступлении соответствующего события. Все события будут возникать в тот момент, когда файл или директория, удовлетворяющая свойствам `Path`, `NotifyFilter` и `Filter`, будет претерпевать изменения.

Свойство	Описание
`Path`	Этот путь должен указывать на местоположение файла или директорию, подлежащие контролю.
`NotifyFilter`	Это свойство представляет собой некоторое сочетание значений перечислимого типа `NotifyFilters`, которое позволяет определить, за наступлением каких именно событий для данных файлов следует наблюдать. Они представляют свойства файлов или папок, за которыми ведется наблюдение. Если какое-либо из указанных свойств оказывается измененным, то генерируется соответствующее событие. Данный перечислимый тип включает в себя следующие значения: `Attributes`, `CreationTime`, `DirectoryName`, `Filename`, `LastAccess`, `LastWrite`, `Security`, `Size`. Обратите внимание на то, что допускается использование различных сочетаний этих значений посредством бинарного оператора `OR`.
`Filter`	Фильтр, определяющий, какие именно файлы подлежат контролю, например, `*.txt`.

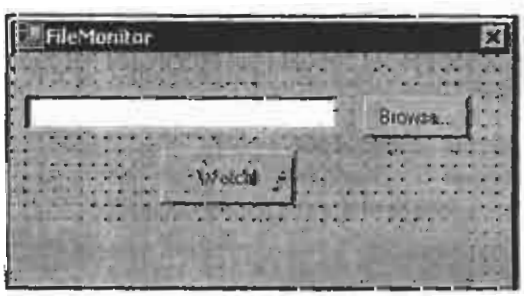
После задания указанных свойств и событий свойству `EnableRaisingEvents` следует присвоить значение `true`, что будет означать начало наблюдения.

Практикум: наблюдение за состоянием файловой системы

1. Создадим более изощренный пример, в котором будет использоваться почти все, с чем мы познакомились в данной главе. Создайте новый проект. Выберите на этот раз C# Windows Application и назовите его `FileWatch`.
2. Задайте свойства различных форм в соответствии с приведенной справа таблицей.

Свойство	Значение
`FormBorderStyle`	`FixedDialog`
`MaximizeBox`	`False`
`MinimizeBox`	`False`
`Size`	302, 160
`StartPosition`	`CenterScreen`
`Text`	`FileMonitor`

3. Используя приведенный ниже список, добавьте в форму необходимые управляющие элементы и задайте их свойства:
   - **Textbox** — `Name`: `txtLocation`; `Location`: 8, 24; `Size`: 184, 20; `Text`: отсутствует
   - **Button** — `Name`: `cmdBrowse`; `Location`: 208, 24; `Size`: 64, 240; `Text`: Browse...
   - **Button** — `Name`: `cmdWatch`; `Location`: 88, 56; `Size`: 80, 32; `Text`: Watch!; `Enabled`: `false` (мы не имеем возможности следить за файлом, до того как он будет задан)



   - **Label** — `Name`: `lblWatch`; `Location`: 8, 104; `Size`: 264, 32; `Text`: отсутствует
   - **OpenFileDialog** — `Name`: `FileDialog`; `Filter`: All Files|*.*

В итоге ваша форма должна будет принять вид, изображенный на рисунке слева. Серое пространство в нижней части формы представляет собой

управляющий элемент Label, в котором будет выводиться текущее состояние вашего приложения.

4. Для того чтобы работа выполнялась корректно, нам потребуется добавить некоторый код. Щелкните правой кнопкой мыши на файле Form1.cs в Solution Explorer, а затем выберите мышью режим просмотра кода. Как всегда, нам придется добавить в проект пространство имен System.IO для того чтобы получить доступ к необходимым классам. Добавьте следующий код в начало файла Form1.cs:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
```

5. Теперь необходимо добавить в класс Form1 класс FileSystemWatcher. После объявлений объектов WinForms с модификатором private добавьте следующий код:

```
private System.Windows.Forms.TextBox txtLocation;
private System.Windows.Forms.Button cmdBrowse;
private System.Windows.Forms.Button cmdWatch;
private System.Windows.Forms.Label lblWatch;
private System.Windows.Forms.OpenFileDialog FileDialog;

// Объект наблюдения за файловой системой
private FileSystemWatcher watcher;
```

6. Теперь требуется внести некоторый код в метод InitializeComponent(). Для того чтобы получить возможность просматривать метод InitializeComponent(), вам придется прежде всего раскрыть более широко область, обозначенную как Windows Form Designer generated code (код, сгенерированный программой Windows Form Designer). Для этого щелкните мышью на маленьком окошке, в котором нарисован знак плюс, в левой части экрана. В результате вы увидите метод InitializeComponent(), в котором уже находится большое количество кода, помещенного туда Visual Studio.NET. Добавьте следующий код в конец метода InitializeComponent(). Этот код необходим для инициализации объекта FileSystemWatcher и привязывает события к методам, к созданию которых мы приступим далее:

```
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 следующие четыре метода после метода InitializeComponent(). Эти методы будут заниматься обработкой событий, создаваемых объектом FileSystemWatcher:

```
// Описание 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();
            lblWatch.Text = "Wrote change event to log";
        }
        catch(IOException ex)
        {
            lblWatch.Text = "Error Writing to log";
        }
    }

    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);
            sw.Close();
            lblWatch.Text = "Wrote renamed event to log";
        }
        catch(IOException ex)
        {
            lblWatch.Text = "Error Writing to log";
        }
    }

    public void OnDelete(object source, FileSystemEventArgs e)
    {
        try
        {
            StreamWriter sw = new StreamWriter("C:/FileLogs/Log.txt",true);
            sw.WriteLine("File: {0} Deleted", e.FullPath);
            sw.Close();
            lblWatch.Text = "Wrote delete event to log";
        }
        catch(IOException ex)
        {
            lblWatch.Text = "Error Writing to log";
        }
    }

    public void OnCreate(object source, FileSystemEventArgs e)
    {
        try
        {
            StreamWriter sw = new StreamWriter("C:/FileLogs/Log.txt",true);
            sw.WriteLine("File: {0} Created", e.FullPath);
            sw.Close();
            lblWatch.Text = "Wrote create event to log";
        }
        catch(IOException ex)
        {
            lblWatch.Text = "Error Writing to log";
        }
    }
```

8. Теперь нужно добавить обработчик событий для кнопки поиска (Browse...). Проще всего этого можно добиться, позволив IDE самостоятельно создать такой обработчик. Перейдите с помощью мыши обратно в окно разработки формы. Нажмите мышью

на кнопке Browse... два раза. Следующий код приводит к открытию диалога Open File, который позволяет пользователю выбрать файл, за которым будет установлено наблюдение. Это реализуется посредством метода ShowDialog():

```
private void cmdBrowse_Click(object sender, System.EventArgs e)
{
   if (FileDialog.ShowDialog() != DialogResult.Cancel )
   {
      txtLocation.Text = FileDialog.FileName;
   }
   cmdWatch.Enabled = true;
}
```

Метод ShowDialog() возвращает значение перечислимого типа DialogResult, которое соответствует тому, каким образом пользователь покинул диалог Open File. У пользователя имеются две возможности: он может нажать либо кнопку OK, либо кнопку Cancel. Необходимо убедиться в том, что пользователь не нажимал кнопку Cancel, поэтому, прежде чем занести выбранный пользователем файл в управляющий элемент Textbox, мы сравниваем полученный по завершении работы данного метода результат со значением перечислимого типа DialogResult.Cancel. После этого свойству Enabled кнопки Watch присваивается значение true — теперь имеется возможность вести наблюдение за выбранным файлом.

9. Переходим к последнему участку кода. Выполните точно такую же процедуру для кнопки Watch. Для запуска объекта FileSystemWatcher добавьте следующий код:

```
private void cmdWatch_Click(object sender, System.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;
   // Начало наблюдения.
   watcher.EnableRaisingEvents = true;
}
```

10. Необходимо убедиться в том, что директория оказывается доступной и в нее можно производить запись данных. Добавьте в конструктор Form1 следующий код, который будет производить проверку на предмет существования соответствующей директории и создавать, если она отсутствует.

```
public Form1()
{
   //
   // Необходимо для поддержки Windows Form Designer
   //
   InitializeComponent();

   DirectoryInfo aDir = new DirectoryInfo("C:/FileLogs");
   if(!aDir.Exists)
      aDir.Create();
}
```

11. Постройте проект. Если процедура построения проекта завершится успешно, то нажмите кнопку **Browse** и выберите какой-либо текстовый файл из имеющихся на вашем компьютере.

12. Для начала наблюдения за файлом нажмите кнопку **Watch**. Единственное видимое изменение, которое произойдет с вашим приложением,— это появление управляющего элемента с надписью, указывающей, за каким именно файлом ведется наблюдение.



13. С помощью Windows Explorer перейдите к тому самому файлу, за которым установлено наблюдение. Откройте этот файл в Notepad, добавьте в него произвольный текст. Сохраните файл.

14. Теперь можно просмотреть регистрационный файл, в котором будут отражены все изменения. Откройте файл `C:\FileLogs\Log.txt` в Notepad. Вы должны увидеть описание произошедших с ним изменений (см. рис. слева).

Как это работает — Наблюдение за состоянием файловой системы

Это очень простое приложение, однако оно позволяет продемонстрировать, как работает объект `FileSystemWatcher`. Если вы зададите для данной директории *.*, то отслеживаться будут все изменения, происходящие в данной директории.

Большая часть кода в приложении касается настройки объекта `FileSystemWatcher`, для того чтобы можно было вести наблюдение в нужном месте:

```
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;
// Начало наблюдения.
watcher.EnableRaisingEvents = true;
```

В этом коде сначала задается путь к директории, в которой будет вестись наблюдение, для чего используется новый для нас объект, с которым нам еще не приходилось сталкиваться,— объект `System.IO.Path`. Это статический класс, очень напоминающий статический объект `File`. Он обладает большим количеством методов, позволяющих выполнять манипуляции и извлекать информацию из строк, представляющих ведущие к файлам пути. Сначала мы используем этот объект для получения имени директории, введенной пользователем в текстовом окне, посредством метода `GetDirectoryName()`.

В следующей строке задается фильтр, который будет использоваться объектом. В качестве фильтра может фигурировать просто имя файла, в этом случае наблюдение будет вестись только за указанным файлом, а может использоваться какая-либо строка вроде `*.txt` — в таком случае наблюдение будет вестись за всеми файлами с расширением `.txt`, находящимися в данной директории. Здесь снова используется статический объект `Path` для извлечения информации из местоположения файла, заданного пользователем.

Свойство `NotifyFilter` представляет собой комбинацию из нескольких перечислимых значений типа `NotifyFilters`, с помощью которых определяется, что входит в понятие изменения. В данном примере мы требуем, что наше приложение должно уведомляться об изменениях времени последней записи в файл, имени файла или размера файла. После внесения изменений в пользовательский интерфейс мы присваиваем свойству `EnableRaisingEvents` значение `true`, что приводит к началу ведения наблюдения.

Однако перед этим нам приходится создать объект и определить обработчики событий:

```
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);
```

Именно таким образом нам удается привязать обработчики событий, отправляемых ведущим наблюдение объектом, к созданным нами частным методам. Именно в этих методах мы принимаем решение о том, каким образом следует обрабатывать наступившее событие. Не забудьте, что уведомление приходит после наступления события.

В методах обработчиков событий, которые используются в настоящем примере, просто производится соответствующая запись в файл регистрации. Реакция на наступившее событие может быть гораздо сложнее — это зависит от самого приложения. Например, если в директорию добавляется новый файл, то его можно перенести куда-либо в другое место или, прочитав его содержимое, запустить какой-либо новый процесс, в котором будет использоваться содержащаяся в нем информация. Наши возможности безграничны!

Асинхронный доступ к файлам

Доступ ко всем файлам на протяжении настоящей главы осуществлялся синхронным образом. Это означает, что выполнение приложения будет приостановлено до тех пор, пока операция чтения или записи не будет завершена. Это оказывается вполне приемлемым для большинства приложений, однако иногда возникает необходимость записать очень большое количество данных, при этом у приложении совершенно отсутствует необходимость дожидаться окончания операции записи. Или, например, требуется считать информацию из файла, находящегося в сети с медленным соединением; в этом случае ваше приложение могло бы продолжать выполнять какую-либо другую работу, параллельно внося соответствующие изменения в управляющие элементы пользовательского интерфейса, отражающие процесс загрузки данных. Такой способ доступа к данным называется асинхронным. Обычно это означает, что приложение может выполнять какие-либо другие действия параллельно с неким процессом, например, с операцией ввода/вывода файла, вместо того чтобы дожидаться окончания этого процесса.

Класс `FileStream` предоставляет несколько методов, предназначенных для осуществления асинхронного доступа к файлам. Это несколько более сложный способ получения доступа к файлам, однако он обладает тем преимуществом, что вы получаете возможность выполнять какие-либо другие действия в то время, пока происходит считывание или запись файла.

Процесс считывания файла начинается с вызова метода `BeginRead()`. Этому методу передаются параметры аналогичные тем, которые передаются основному

методу `Read()`, плюс дополнительные параметры, необходимые для осуществления асинхронного процесса.

```
IAsyncResult BeginRead(byte[] array, int offset, int numByte, AsyncCallack
userCallback, object stateObject);
```

Первые три параметра абсолютно идентичны параметрам метода `Read()`. Параметр `userCallback` — это специально создаваемый метод, с помощью которого .NET Framework возвращает приложению сообщение о том, что процедура чтения завершена. Этот параметр будет рассматриваться позднее. Поскольку этот параметр не является обязательным, то вместо него можно передавать `null`, что мы и сделаем в нашем первом примере. Последний параметр — это задаваемый пользователем объект состояния, который используется совместно с методом, возвращающим сообщение, для разделения нескольких асинхронных операций. Поскольку существует возможность использовать один и тот же метод сразу для нескольких операций, то можно использовать данный параметр для передачи какой-либо строки или данных, позволяющих идентифицировать данную асинхронную операцию. Поскольку метод для возвращения сообщения нами не используется, вместо этого параметра также передается `null`.

Возвращаемый объект `IAsyncResult` является абстрактным представлением процесса чтения. В нем содержится большое количество свойств и методов, которые позволяют получать информацию о состоянии процесса, а также определять, что данный процесс закончился.

Для завершения процесса чтения вызывается метод `EndRead()`; ему передается объект `IAsyncResult`, который был возвращен методом `BeginRead()`. Познакомимся с тем, как все это работает, с помощью примера.

Практикум: асинхронное считывание файлов

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его `AsyncRead`.

2. Добавьте следующие строки в начало файла `Class1.cs`:

```
using System;
using System.IO;
using System.Text;
```

3. Добавьте следующий код в метод `Main()`:

```
static void Main(string[] args)
{
   byte[] byData = new byte[100];
   char[] charData = new Char[100];

   try
   {
      FileStream aFile = new FileStream("../../Class1.cs ",FileMode.Open);
      aFile.Seek(0,SeekOrigin.Begin);

      // Начало процедуры асинхронного чтения из файла
      System.IAsyncResult asResult = aFile.BeginRead(byData,0,100,null,null);

      // Выполнение других действий параллельно с чтением данных
      while (!asResult.IsCompleted)
      {
         Console.WriteLine("Reading from file...");
      }
```

```
            aFile.EndRead(asResult);
            Decoder d = Encoding.ASCII.GetDecoder();
            d.GetChars(byData,0,byData.Length,charData,0);
            Console.WriteLine(charData);

            Console.ReadLine();
            return;
         }
         catch(IOException e)
         {
            Console.WriteLine("An IO exception has been thrown!");
            Console.WriteLine(e.ToString());
            Console.ReadLine();
            return;
         }
      }
```



4. Запустите приложение — вы сможете увидеть, что начало файла `Class1.cs` будет записано в файл после нескольких строк `"Reading from file..."`, как показано на рисунке слева.

Как это работает — Асинхронное считывание файлов

В данном примере происходит достаточно много интересного. Прежде всего мы запускаем процедуру считывания файла с помощью следующей строки:

```
System.IAsyncResult asResult = aFile.BeginRead(byData,0,100,null,null);
```

Это приводит к тому, что приложению возвращается объект `IAsyncResult`, с помощью которого можно запрашивать информацию о том, завершилась ли на данный момент процедура считывания или нет. Пока процедура не будет завершена, можно выполнять какие-либо другие действия на фоне выполняющейся процедуры считывания файла:

```
while (!asResult.IsCompleted)
{
   Console.WriteLine("Reading from file|");
}
```

В этом примере не выполняется никаких действий, кроме простого вывода на консоль сообщения о том, что программа находится в состоянии ожидания, однако приложение может выполнять параллельно какие-нибудь гораздо более сложные действия. Однако наши возможности по-прежнему сильно ограничены, поскольку нам приходится постоянно крутиться в цикле `while`, ожидая завершения выполнения метода `Read()`. К счастью, существует другой способ — как раз здесь в дело вступают остальные параметры метода `BeginRead`.

Асинхронный доступ к файлам с использованием событий

Другой способ осуществления доступа к файлам в асинхронном режиме включает использование метода, возвращающего сообщение, который позволяет .NET Framework предать приложению информацию о том, что операция завершена.

И хотя приложение в этот момент может заниматься чем-либо другим, оно немедленно уведомляется о завершении операции чтения.

Для этого необходимо создать метод, которому будет передаваться объект `IAsyncResult`. Этот метод может являться составной частью как текущего класса, так и какого-либо другого. В настоящем примере мы включили его как член самого класса, однако это не является обязательным требованием.

Практикум: асинхронное считывание файлов с использованием событий

1. Создайте новый проект. Выберите пункт меню C# Console Application и назовите его `AsyncReadEvents`.

2. Измените имя класса с `Class1` на `AsyncRead`.
   Программа должна выглядеть следующим образом:

```
using System;

namespace AsyncReadEvents
{
    /// <summary>
    /// Краткое описание класса Class1.
    /// </summary>
    class AsyncRead
    {
        static void Main(string[] args)
        {
            //
            // Что сделать: добавить код для запуска приложения
            //
        }
    }
}
```

3. Включите следующие строки кода в начало файла `Class1.cs`:

```
using System;
using System.IO;
using System.Text;
```

4. В этом примере мы поступим несколько по-другому. Добавьте следующие переменные в класс `AsynchRead`:

```
public class AsynchRead
{
    byte[] byData;
    char[] charData;
```

5. Включите следующий код конструктора класса `AsynchRead` непосредственно вслед за только что определенными переменными:

```
public AsynchRead()
{
    byData = new byte[100];
    charData = new Char[100];

    try
    {
        FileStream aFile = new FileStream("../../Class1.cs",FileMode.Open);
        System.AsyncCallback cb = new AsyncCallback(this.HandleRead);
        System.IAsyncResult aResult = aFile.BeginRead(byData,0,100,cb,"Read Class1.cs");
    }
```

```
        catch(IOException e)
        {
            Console.WriteLine("An IO exception has been thrown!");
            Console.WriteLine(e.ToString());
            Console.ReadLine();
            return;
        }
    }
```

6. Теперь добавим метод, возвращающий сообщение,— `HandleRead()`, который был зарегистрирован нами перед этим в методе `BeginRead()`. Добавьте этот код в произвольное место класса `AsynchRead`.

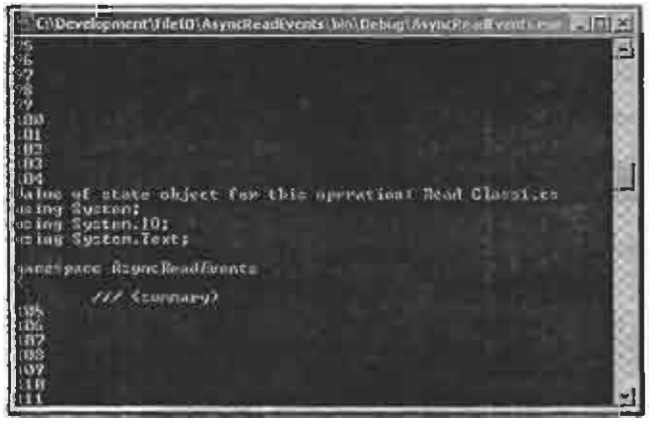
```
private void HandleRead(IAsyncResult ar)
{
    Decoder d = Encoding.UTF8.GetDecoder();
    d.GetChars(byData, 0, byData.Length, charData, 0);
    Console.WriteLine("Value of state object for this operation: {0}",
                      ar.AsyncState);
    Console.WriteLine(charData);
}
```

7. В заключение добавьте приведенный ниже код в метод `Main()`. В нем создается новый экземпляр объекта `AsynchRead`, что позволяет запустить приложение.

```
static void Main(string[] args)
{
    AsynchRead aClass = new AsynchRead();

    for(int x=0;x<200;x++)
    {
        Console.WriteLine(x);
    }

    Console.ReadLine();
    return;
}
```



8. Запустите приложение. Вы увидите выходной поток, подобный представленному на рисунке слева.

Как это работает — Асинхронное считывание файлов с использованием событий

Этот пример обладает фундаментальными отличиями от предшествующего. В нем метод `Main()` создает экземпляр объекта `Class1`, а затем продолжает выполнять свою собственную работу — вывод порядковых чисел на консоль; весь смысл этого заключается в том, что теперь этому методу не приходится отслеживать процедуру считывания, и он может выполнять какие-либо другие действия, совершенно от этой процедуры не зависящие.

Считывание файла в асинхронном режиме запускается в конструкторе класса `Class1`, при этом в системе регистрируется метод, возвращающий сообщение.

Следующие после открытия объекта FileStream две строки кода выполняют все для этого необходимое:

```
System.AsyncCallback cb = new AsyncCallback(this.HandleRead);
System.IAsyncResult aResult = aFile.BeginRead(byData,0,100,cb,"Read Class1.cs");
```

В первой строке происходит создание объекта AsyncCallback, который передается в качестве четвертого параметра методу BeginRead(). Для этого создается совершенно новый объект AsyncCallback, передавая адрес метода, который будет принимать объект IAsyncResult в качестве своего единственного параметра. В следующей строке осуществляется вызов метода BeginRead(), которому передаются стандартные данные, с которыми мы познакомились ранее, и, кроме того, объект AsyncCallback и строка "Read Class1.cs". Это — задаваемый пользователем объект состояния, который будет использоваться нами позднее — в методе, возвращающем сообщение. Это не имеет никакого смысла для данного сценария, поскольку мы осуществляем только один запрос на асинхронное считывание, но если бы таких запросов было несколько, то нам потребовался бы способ отличать один от другого.

В методе HandleRead() не происходит ничего такого, чего бы нам не встречалось ранее. Он просто перекодирует данные и выводит символьный массив на консоль. В нем также выводится значение свойства AsyncState объекта IAsyncResult.

```
private void HandleRead(IAsyncResult ar)
{
   Decoder d = Encoding.UTF8.GetDecoder();
   d.GetChars(byData, 0, byData.Length, charData, 0);
   Console.WriteLine("Value of state object for this operation: {0}",
                     ar.AsyncState);
   Console.WriteLine(charData);
}
```

Наиболее интересен тот факт, что данный метод не вызывается приложением. Он вызывается системой .NET Framework в тот момент, когда завершается процедура считывания файла. Это позволяет приложению продолжать обработку какой-либо иной информации, пока выполняется относительно медленная процедура считывания файла.

Асинхронная запись файлов

До сих пор мы рассматривали только чтение файлов в асинхронном режиме, однако все эти способы вполне применимы и при записи в файл в асинхронном режиме. В этом случае также имеется возможность выбора — использовать или не использовать события, при этом преимущества обоих способов остаются теми же. Поскольку мы уже неоднократно сталкивались с процедурой записи файлов, а принципы доступа к файлам в асинхронном режиме оказываются идентичными, в конце данной главы дается упражнение на асинхронную запись файлов.

Итоги

Из настоящей главы мы узнали, что такое потоки и почему они используются в .NET Framework для осуществления доступа к файлам и другим последовательным устройствам. Мы рассмотрели основные классы, находящиеся в пространстве имен System.IO: File, FileInfo и FileStream, а также обсудили вопрос, для чего они могут использоваться. Мы также изучили классы StreamReader и StreamWriter

и то, насколько они оказываются полезными при осуществлении записи в поток и чтении из потока. Мы узнали, каким образом может осуществляться чтение и запись в файлы произвольного доступа посредством класса `FileStream`. Наконец, мы рассмотрели более сложные решения, которые можно строить с использованием .NET Framework. Мы построили приложение, которое позволяет вести наблюдение за файлами и директориями с помощью класса `FileSystemWatcher`.

В результате нами было рассмотрено:

- Открытие файлов
- Чтение из файлов
- Запись в файлы
- Различия между классами `StreamWriter` и `StreamReader`, с одной стороны, и классом `FileStream`, с другой.
- Преимущества и недостатки асинхронного доступа к файлам
- Наблюдение за файловой системой с использованием класса `FileSystemWatcher`

Упражнения

1. Как называется пространство имен, которое должно импортироваться для того, чтобы приложение могло работать с файлами?
2. В каких случаях для записи в файл следует использовать объект `FileStream` вместо объекта `StreamWriter`?
3. Какие методы класса `StreamWriter` позволяют производить считывание данных из файлов, и что конкретно делает каждый из них?
4. Какие события может генерировать класс `FileSystemWatcher`, и для чего они предназначены?
5. Модифицируйте приложение FileWatch, созданное нами в настоящей главе. Предусмотрите в нем возможность включать и отключать процедуру наблюдения без необходимости выходить из приложения.
6. Создайте приложение, которое производит запись данных в файл в асинхронном режиме. Создайте произвольным образом большой массив данных и запишите его в файл.

21

Модули .NET

Модулем в .NET называется выполняемая программа (или ее некоторая часть), поставляемая в виде отдельного блока. Модули — это средства для создания пакетов программ на C# для выполнения и распространения. Когда вы строите программу на C#, то получаемый в результате файл с расширением `.exe` представляет собой не что иное как модуль. Если вы строите библиотеку классов, то получаемый в результате файл DLL (Dynamic Link Library — динамически загружаемая библиотека) также представляет собой модуль.

Весь код, находящийся в некотором модуле, строится, распространяется и получает номер версии как единое целое. Модуль экспортирует общие классы, свойства и методы, доступные для других программ. Все объекты, описанные как частные, остаются внутренними для модуля. Для получения более подробной информации следует продолжить чтение книги. Для получения дополнительной информации вам можно осуществить поиск по доступной в режиме онлайн документации .NET Framework для термина "Assembly" (модуль).

Хотя все программы на C# упаковываются в виде модулей, многие возможности модулей разработаны таким образом, чтобы упростить распространение программ в виде специального класса, который называется **компонентом**. Понимание компонентов оказывается существенным для понимания тех преимуществ, которое дает использование модулей.

Компоненты

Компонентом называется некоторая подпрограмма или часть программы, которая предназначается для использования другой программой. Кроме того, компонент представляет собой блок в **двоичном виде** (т. е. выполняемый код, а не исходный текст), который может использоваться другими программами без необходимости повторной компиляции как самого компонента, так и программы, использующей этот компонент. Это означает, что от сторонних производителей не требуется предоставлять исходный код для своих компонентов.

Слово "компонент" может быть определено различными способами. В самом общем смысле этот термин описывает произвольную подпрограмму, представленную в двоичном виде; при таком использовании этого термина любая DLL по определению является компонентом, поскольку она представляет собой подпрограмму, содержащую выполняемый код.

Более строгое определение налагает требование, гласящее, что компонент должен обладать средствами, которые позволяют ему объявлять о собственном содержимом другим программам. В рамках .NET все модули обладают такими средствами объявления.

Наиболее строгое определение компонента требует, чтобы компонент обладал определенными интерфейсами, позволяющими освобождать не используемые более системные ресурсы, и поддерживал интеграцию с инструментами, использующимися для разработки. В рамках .NET Framework компонент в таком строгом смысле этого слова должен обладать интерфейсом `System.ComponentModel.IComponent`, который обеспечивает реализацию указанных возможностей.

Для того чтобы понять преимущества модулей, мы будем пользоваться менее строгим определением, которое включает в себя налагаемое на компонент требование объявления о собственном содержимом.

Преимущества компонентов

Компоненты обеспечивают улучшенные возможности повторного использования и гибкость, а также упрощают распространение подпрограмм. Повторное использование программ, представленных в двоичном виде, экономит время и повышает надежность.

Представьте себе класс `Shapes` (фигуры), в котором находятся различные объекты, представляющие окружности, треугольники и другие геометрические фигуры. В нем могут содержаться методы, предназначенные для подсчета площади фигур или для выполнения каких-либо других операций над фигурами. Класс `Shapes` может использоваться программами из самых разных областей: рисования/создания графических объектов, разработки технических, архитектурных и строительных чертежей, проектирования с использованием компьютера, а также игровыми и другими программами.

Было бы замечательно, если бы можно было описать подпрограммы, предназначенные для рисования и выполнения различных манипуляций над фигурами, только один раз, а потом использовать их во всех перечисленных программах. Именно в этом и заключается преимущество повторно используемого кода.

Многократное использование без необходимости каждый раз проходить компиляцию и настройку ссылок для каждой из использующих этот класс программ позволило бы сэкономить большое количество времени и привело бы к увеличению надежности (поскольку при выполнении процедур повторной компиляции и настройки ссылок всегда существует вероятность внесения ошибок).

Более того, возможно какой-то другой программист или компания уже создали некоторый компонент класса `Shapes`, который выполняет необходимые вам операции; в этом случае вы сможете воспользоваться этим компонентом (переписав и/или купив его), и вам не придется создавать его самому. Раз существует возможность обеспечения общего доступа к компонентам, представленным в двоичном виде, то не придется беспокоиться о том, какой язык программирования был использован при создании данного компонента. .NET Framework и ее модули позволяют пользоваться всеми этими преимуществами.

Краткая история компонентов

Для того чтобы обладать возможностью многократно использовать компоненты, представленные в двоичном виде, должен существовать некий стандарт, определяющий способ, который будет использоваться при именовании и применении объектов на двоичном уровне. Такой стандарт для продуктов, работающих в среде Microsoft, со временем претерпевал определенные изменения.

В Microsoft Windows было введено понятие DLL, что позволило одной или нескольким программам использовать блок кода, хранящийся в виде отдельного файла.

Все это работало на довольно примитивном уровне — только в том случае, если все программы были написаны на одном и том же языке программирования (обычно на C). При этом программы должны были обладать большим количеством предварительной информации об используемой DLL, а сами DLL не позволяли программам обмениваться своей информацией.

Для обмена данными между программами была разработана система DDE (Dynamic Data Exchange — динамический обмен данными). В ней определялся формат и механизм передачи данных от одной программы другой, но этой системе не хватало гибкости. Вслед за ней появилась система OLE 1.0 (Object Linking and Embedding — загрузка и встраивание объектов), которая обеспечивала реальную возможность документу из одной программы (например, Excel) входить в состав другого документа (например, документа Word). Это уже в некотором смысле напоминало компоненты, хотя на самом деле OLE 1.0 не представляла собой настоящий многофункциональный стандарт компонента.

В середине 90-х компания Microsoft описала свой первый стандарт компонента в стандарте COM (Component Object Model — объектная модель компонента), реализованного в среде Windows. OLE версии 2 и множество появившихся впоследствии технологий были построены на основе COM. В DCOM (Distributed COM — распределенная COM) была введена возможность, позволяющая COM-компонентам взаимодействовать в сети, а в COM+ добавились службы, которые могли вызываться компонентами для обеспечения высокого быстродействия в среде многозвенных приложений.

COM работает неплохо, однако она оказывается очень сложной для изучения и использования (в особенности, если она используется из C++). COM налагает требование включать информацию о компонентах в системный реестр Windows, что значительно усложняет инсталляцию компонентов и затрудняет их удаление.

Изначально COM разрабатывалась для использования с C/C++; она была расширена таким образом, чтобы Visual Basic мог использовать ее ("Automation", автоматизация), и все это на самом деле работало замечательно, хотя в результате программистам, работающим на C/C++, стало даже труднее добиваться совместимости компонентов с Visual Basic (так, например, по-прежнему отсутствовала возможность осуществлять наследование от класса, описанного на другом языке программирования).

Ко всему прочему, по мере того как пользователи начали инсталлировать различные версии DLL и COM-компонентов, созданные компанией Microsoft и другими производителями, стали возникать проблемы, поскольку разные программы использовали разные версии одной и той же DLL общего доступа. Оказалось, что одной программе не составляет никакого труда инсталлировать другую версию той DLL, которая уже используется какой-либо иной программой, что приводило к нарушению работы этой программы (это явление получило название "ад DLL"). Необходимость отслеживать всю информацию о различных DLL, инсталлированных на одной системе, существенно затрудняло процесс обновления и сопровождения компонентов.

В модели программирования .NET определяется новый стандарт, призванный решить эти проблемы.

Самоописание

Наиболее важным аспектом модулей .NET, отличающим их от всех своих предшественников, является их полная **самоописательность**. Описание содержится внутри самого модуля, поэтому системе или вызывающей программе не требуется

обращаться к реестру или куда-либо еще за информацией относительно содержащихся в модуле объектов.

Самоописание в модулях .NET выходит далеко за рамки простого описания имен объектов и методов, а также типов данных, использующихся в качестве параметров; в модулях кроме того содержится информация о том, какова версия данного объекта (представьте себе Shapes 1.0, за которым следует Shapes 1.1 или Shapes 2.0), и обеспечивается безопасное использование содержащихся в них объектов. Вся эта информация содержится непосредственно внутри самого модуля, поэтому нет никакой необходимости осуществлять поиск этой информации где-либо еще. В результате этого инсталляция .NET-компонентов становится намного более простой и понятной процедурой по сравнению со всеми предшествующими технологиями Windows. Она может быть сведена к простому копированию модулей на диск в системе, для которой они предназначены.

Модули .NET и библиотека классов .NET Framework

Каждой программой в .NET, включая программы на C#, широко используется библиотека классов .NET Framework. Эти классы используются каждый раз, когда вызывается какой-либо метод из пространства имен `System` с помощью директивы `using System` — все пространства имен `System` (`System.Data`, `System.Drawing` и т. д.) входят в состав библиотеки классов .NET Framework.

Каждый класс, входящий в состав этой библиотеки, представляет собой часть самоописывающегося модуля. Классы, предназначенные, например, для рисования, содержатся в модуле `System.Drawing.dll`. Если, работая в Visual Studio.NET, вы включите ссылку на этот модуль, то компилятор включит эту ссылку в модуль, который он будет строить для вашей программы. Система выполнения программ — CLR — считывает метаданные из модуля вашей программы, для того чтобы извлечь информацию относительно тех модулей, которые ей требуются, а затем осуществляет поиск и загрузку этих модулей для использования их вашей программой. Модули, на которые ссылается ваша программа, могут, в свою очередь, ссылаться на другие модули, поэтому даже самая простая программа, в которой используется единственная директива `using`, может ссылаться на несколько различных модулей. Самоописание каждого модуля отслеживает все эти ссылки, поэтому от вас не требуется даже знать о них.

> *Здесь необходимо внести определенную ясность, с тем чтобы избежать путаницы в дальнейшем. Пространства имен и модули не всегда в точности соответствуют друг другу. Так, например, в модуле `System.Data.dll` реализованы некоторые программы, одновременно принадлежащие пространствам имен `System.Data` и `System.Xml`, хотя другие программы из пространства имен `System.Xml` реализованы в модуле `System.Xml.dll`.*
> *Внутри программы с помощью директивы using вы ссылаетесь на пространство имен; ссылки в проекте Visual Studio.NET определяют конкретные используемые модули.*

Одновременное использование разных языков программирования

Дополнительное преимущество модулей и всей системы .NET заключается в возможности одновременно использовать различные языки программирования, поскольку компоненты могут вызываться из любого языка программирования,

включенного в .NET, независимо от того, на каком языке этот компонент был изначально написан.

В .NET имеется несколько особенностей, которые делают возможным одновременное использование нескольких различных языков программирования:

- Common Language Runtime (CLR — единый язык выполнения программ), который управляет выполнением всех модулей в .NET.
- MSIL (Microsoft Intermediate Language — промежуточный язык компании Microsoft), который генерируется компиляторами со всех языков программирования, включенных в .NET. Он представляет собой единый стандарт двоичного кода, генерируемого компиляторами, и именно он выполняется CLR. В CLR также определяется формат, в котором хранятся метаданные модулей, а это означает, что для хранения метаданных во всех модулях используется единый формат независимо от того, на каком языке этот модуль разрабатывался.
- Common Language Specification (CLS — единая спецификация языков программирования), которая устроена таким образом, что программы, написанные на C#, Visual Basic, C++ или любых других CLS-совместимых языках программирования .NET, могут получать общий доступ к компонентам с наследованием, выходящим за рамки отдельного языка. В CLS определяются возможности, которые должны поддерживаться языками для обеспечения их совместного использования друг с другом. Существует возможность использовать и другие особенности языков программирования, не входящие в CLS, но в этом случае отсутствует гарантия того, что эти особенности поддерживаются в других языках.
- Common Type System (CTS — единая система типов), которая определяет базовые типы, используемые в языках программирования .NET и правила определения программистами своих собственных классов. Это позволяет избегать реализации типа `string` недопустимым образом.

Следуя CLS-спецификации, вы можете написать на C# некоторый компонент, а модуль, в котором этот компонент содержится, может быть использован из программы, написанной на другом языке программирования .NET, таком как Visual Basic.NET, поскольку оба компонента — и C#, и VB.NET — будут выполняться CLR. В программах, написанных на C#, могут использоваться компоненты, написанные на VB.NET, Managed C++ и т. д. На уровне модулей все классы, объекты и типы данных могут совместно использоваться всеми языками программирования .NET, поэтому классы могут наследоваться, а компоненты полноценно использоваться независимо от того, на каком языке программирования они написаны.

Самоописывающиеся модули для COM и другого унаследованного кода

.NET Framework также допускает использование с C# и другими языками программирования .NET компонентов и библиотек, написанных с использованием COM и других "унаследованных" технологий.

Этот механизм также предполагает использование самоописывающихся модулей; на самом деле при этом происходит создание модуля, который **охватывает** доставшийся по наследству код; такой модуль представляет .NET собственное описание, преобразовывает типы данных COM в типы данных .NET и обеспечивает возможность вызывать унаследованный код из языков программирования .NET и наоборот.

В Visual Studio.NET происходит автоматическое создание охватывающего модуля в тот момент, когда вы включаете ссылку на COM-компонент (посредством вкладки COM в диалоге Add Reference).

На приведенной ниже диаграмме изображен такой охватывающий модуль (который также называют вызываемой оболочкой периода выполнения). Вызовы такого модуля, осуществляемые .NET-клиентом, проходят через оболочку, прежде чем достигают самого COM-компонента; с точки зрения модуля .NET охватывающий модуль представляет собой обычный компонент.



Структура модулей

Отдельные части модуля обеспечивают для программ в .NET возможность находить друг друга и раскрывать ссылки программ и компонентов друг на друга.

В модуле выполняемый код программы или библиотеки классов хранится совместно с **метаданными** (данными, описывающими другие данные), которые позволяют другим программам находить классы, методы и свойства объектов, определяемые в рамках данного модуля. Метаданные выступают в двух ипостасях: в виде оглавления, в котором описывается содержимое данного модуля, и в виде библиографии, где описываются ссылки на различные данные, находящиеся вне модуля. Давайте рассмотрим этот вопрос более подробно.

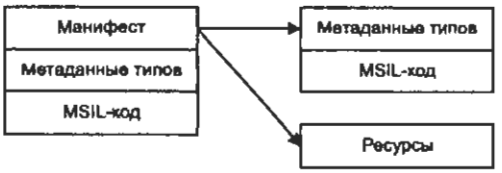
Модули .NET, состоящие из одного файла, обладают общим форматом, представленном на рисунке справа.



В каждом модуле содержится **маиифест**, где описывается содержимое данного модуля. Манифест также называется **метаданными модуля**, поскольку он описывает собственный модуль — что в нем содержится, на какие другие модули он ссылается и т. д. Манифест модуля будет подробно рассматриваться ниже в этой главе, когда будет изучаться содержимое созданного модуля. Предыдущие технологии с использованием компонентов, такие как COM, не имели встроенных объектов, аналогичных манифесту; манифест — это ядро самоописания, содержащееся в модулях .NET.

Система выполнения программ .NET использует манифест, содержащийся в модуле программы, для раскрытия ссылок на другие модули, имеющиеся в системе, например, на библиотеку `System.Console`, в которой содержится метод `WriteLine()`, позволяющий вывести "Hello, World!".

За манифестом следуют **метаданные типов** — описание классов, свойств, методов и т. п., которые содержатся в модуле наряду с описанием типов данных, используемых ими параметров и возвращаемых значений. Далее следует собственно двоичный код для каждого типа, хранящийся в виде машиннонезависимого кода на языке Microsoft Intermediate Language (MSIL — промежуточный язык компании Microsoft). Наконец, следует еще одна часть модуля — **ресурсы**, представляющие собой невыполняемые части программы (представленные в виде файлов с расширением `.Resources`), например, файлы с рисунками, иконками или сообщениями.



Хотя часто модуль состоит только из одного файла, он также может быть составлен из нескольких файлов, как показано на рисунке слева.

С точки зрения системы выполнения программ .NET модуль, состоящий из нескольких файлов, представляет собой единую логическую DLL или EXE, которая состоит из нескольких файлов. Только один файл содержит манифест, в котором помещаются ссылки на остальные файлы, входящие в состав данного модуля. Файлы, в которых хранится выполняемый код, называются **отсеками**, в них находятся метаданные типов и MSIL-код. Кроме того, в наличии могут быть еще файлы ресурсов, содержащие невыполняемый код.

Модули, состоящие из нескольких файлов, обычно используются только в некоторых очень сложных приложениях. Отдельный отсек или ресурс загружается только в тот момент, когда он реально начинает выполняться или использоваться, поэтому, когда редко используемые отсеки и ресурсы хранятся в виде отдельных файлов, удается сократить время, затрачиваемое на загрузку, и занимаемую память. Например, некое приложение, используемое в различных странах, может обладать какими-либо отсеками или ресурсами, написанными на различных национальных языках; такие отсеки или ресурсы должны выделяться в отдельные файлы, с тем чтобы в действительности в память загружался только отсек с фактически используемым локальным языком.

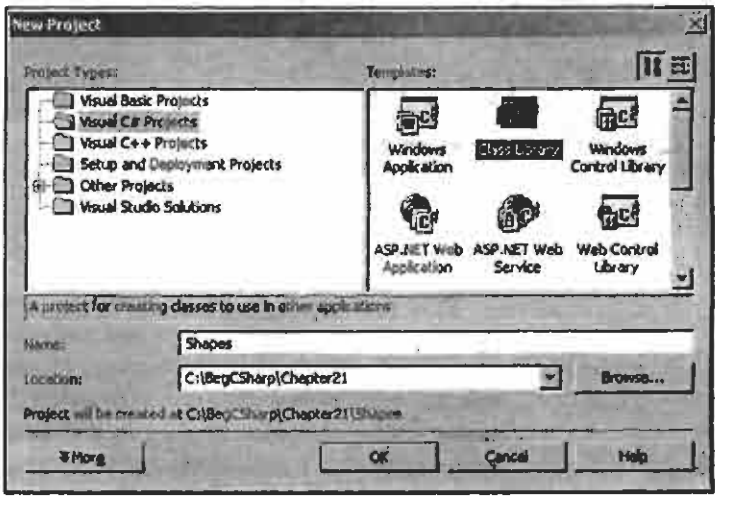
Создание модулей

Теперь попробуем создать модуль и познакомимся с его свойствами. Сначала потребуется создать простую библиотеку классов на C#, на которую мы будем ссылаться на протяжении оставшейся части данной главы. Мы создадим упрощенную версию компонента `Shapes`, который был упомянут в начале главы.

Создание библиотеки классов

Воспользуйтесь Visual Studio.NET и создайте проект библиотеки классов `Shapes` (см. рис. справа).

Переименуйте файл `Class1.cs`, создаваемый по умолчанию, в `shapes.cs` и введите в него исходный код, приведенный ниже. Двоичный файл, который будет создан на основе файла `shapes.cs`, получит название `shapes.dll` и будет использоваться в качестве примера модуля.



Исходный код для Shapes

Исходный код на C# для модуля `shapes.dll` имеет следующий вид:

```
namespace Shapes
{
   public class Circle
   {
      double Radius;
```

```
        public Circle()
        {
            Radius = 0;
        }

        public Circle(double givenRadius)
        {
            Radius = givenRadius;
        }

        public double Area()
        {
            // площадь равняется числу π, умноженному на квадрат радиуса
            return System.Math.PI * (Radius * Radius);
        }
    }

    public class Triangle
    {
        double Base;            (Заглавная буква B используется здесь с целью избежать
                                 путаницы с ключевым словом base.— Прим. пер.)
        double Height;

        public Triangle()
        {
            Base = 0;
            Height = 0;
        }

        public Triangle(double givenBase, double givenHeight)
        {
            Base = givenBase;
            Height = givenHeight;
        }

        public double Area()
        {
            return 0.5F * Base * Height;        // площадь равняется одной второй
                                                // произведения основания на высоту
        }
    }
}
```

Этот код является очень простым, и в нем не реализованы все возможности, которые могут потребоваться при работе с геометрическими фигурами, но для наших целей этого вполне достаточно. Единственная хитрость заключается в том, что каждый класс — и `Circle` (окружность), и `Triangle` (треугольник) — обладает двумя конструкторами, одному из которых вообще не требуются параметры, а другому параметры необходимы для инициализации экземпляров переменных. Мы вернемся к этому вопросу позднее, при изучении модулей.

Компиляция библиотеки классов

При использовании Visual Studio.NET для построения проекта shapes следует выбрать опцию **Build | Build** (*Ctrl-Shift-B*).

Если вы используете компилятор C# из командной строки, то файл `shapes.cs` следует откомпилировать с помощью следующей команды:

```
csc /target:library shapes.cs
```

Это приведет к созданию файла с именем `shapes.dll` в текущей директории. Опция `/target:library` указывает компилятору на необходимость создания требуемого модуля в виде DLL, а не в виде выполняемого файла (с расширением `.exe`).

> *Если возникли проблемы с выполнением команды* `csc` *из командной строки, то это может являться следствием неверного значения переменной среды* PATH. *Правильная настройка этой переменной описывается в приложении А.*

Просмотр содержимого модуля

Просмотрим содержимое только что созданного модуля `shapes.dll` с использованием описания, содержащегося внутри него. В качестве инструмента для просмотра содержимого модуля будет использоваться `ildasm` (Intermediate Language DisASSeMbler tool — инструмент для дезассемблирования промежуточного языка). Это очень удобный инструмент, предназначенный для просмотра внутренней структуры модулей в доступном представлении. Однако, вам вряд ли придется использовать его при повседневной разработке программ на C#, поэтому запоминать детали его использования совершенно необязательно.

Включение ILDASM в качестве внешнего инструмента в VS

`ildasm` представляет собой внешний инструмент, который может быть включен в состав Visual Studio.NET. Для этого потребуется перейти в меню **Tools | External Tools**, которое имеется в Visual Studio.NET. щелкните мышью кнопку **Add** в этом диалоговом окне. Вы увидите строку **[New Tool 1]** в списке в окне **Menu Contents** и текстовое окно **Title**, предназначенное для ввода текста. Введите в окно **Title** название **ildasm**, затем нажмите на кнопку поиска (...), расположенную справа от окна **Command**. В появившемся диалоговом окне **Open** пройдите по следующему пути:



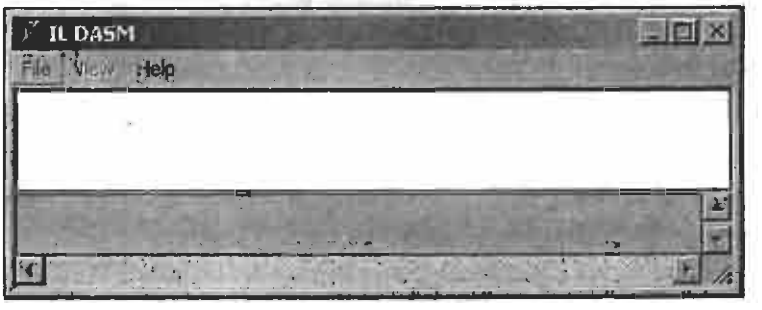

```
C:\Program Files\Microsoft.Net\FrameworkSDK\Bin
```

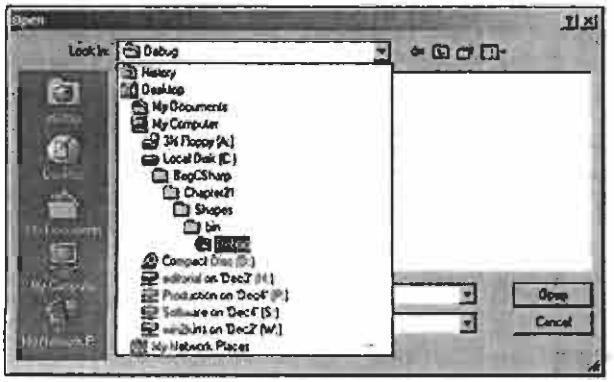
Щелкните мышью на файле `ildasm.exe`, расположенном в директории `bin`, а затем щелкните на **Open**. После этого `ildasm` появится в списке **Menu Contents** (см. рис. слева).

Теперь щелкните мышью **OK**, и `ildasm` появится как выбранный вариант в меню **Tools** в Visual Studio.NET.

Использование ILDASM

После того как `ildasm` включен в Visual Studio.NET, к нему можно обратиться, выбрав пункт меню **Tools | ildasm**. `ildasm` будет выведен в виде отдельного окна, показанного на рисунке справа.

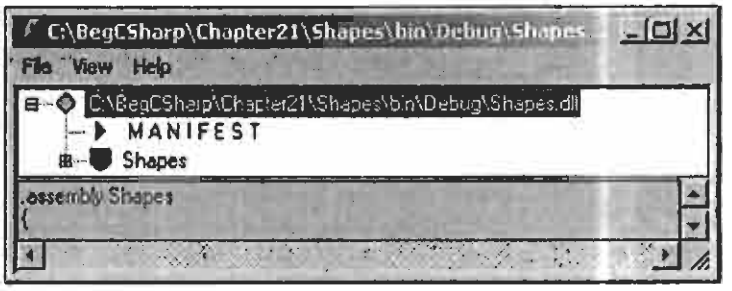


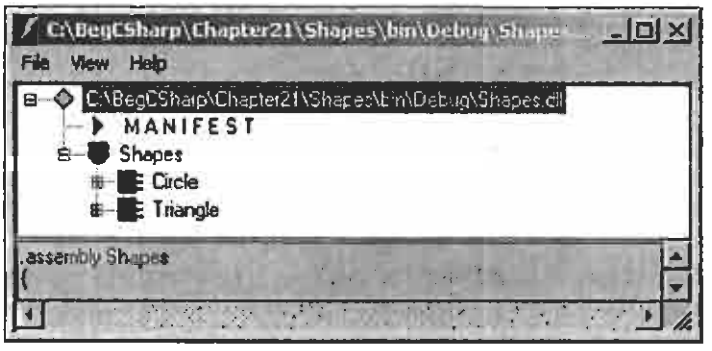


Воспользуйтесь меню File | Open для поиска директории, в которой находится `shapes.dll`, и откройте ее. Если для компиляции программы использовался компилятор командной строки, то файл `shapes.dll` будет располагаться в той же директории, что и исходный файл `shapes.cs`. Если вы использовали для этих целей Visual Studio.NET, то файл `shapes.dll` будет помещен в поддиректорию `bin` вашего проекта, возможно, в поддиректорию `Debug`, если вы строили конфигурацию по умолчанию, как показано на рисунке слева.

После того как вы нашли файл `shapes.dll`, щелкните мышью на кнопке Open — в главном окне `ildasm` появится представление модуля, как показано на рисунке справа.

В данном окне можно увидеть манифест и класс `Shapes`. Манифест — это информация названного модуля.





Иконка Shapes представляет пространство имен `Shapes`; ее изображение извлекается из метаданных типов, содержащихся в данном модуле. Раскройте просматриваемое дерево, щелкнув мышью на знаке +, расположенном перед иконкой `Shapes` (см. рис. слева). Теперь мы можем увидеть оба класса, описанные в исходном файле, — `Circle` и `Triangle`.

Раскройте изображение каждого из классов, щелкнув мышью на значках +, расположенных соответственно перед `Circle` и перед `Triangle`.

Теперь можно увидеть то, что соответствует созданному коду. Представлены переменные экземпляров классов — `radius` для класса `Circle` и `Base` и `Height` для класса `Triangle`, а также метод `Area` для обоих классов. Можно видеть строки, которые окажутся добавленными в исходный код, например, начинающиеся с `.ctor` и `.class` (см. рис. справа).

При более внимательном рассмотрении, можно обнаружить, что строки, начинающиеся с `.ctor`, на деле соответствуют конструкторам, которые были определены для классов `Circle` и `Triangle`.



Для каждого из них предусмотрено по два конструктора, одному из которых не требуется никаких параметров, а второму должны передаваться параметры, необходимые для инициализации экземпляров переменных. Для класса `Circle` при выводе содержимого модуля инструментом `ildasm` конструктору с параметрами передается единственное значение типа `float`, которое соответствует следующей строке:

```
.ctor void(float32)
```

Для класса `Triangle` конструктору, использующему параметры, должны передаваться два параметра типа `float`. Это соответствует следующей строке:

```
.ctor void(float32, float32)
```

Строки, начинающиеся с точки, служат директивами в языке MSIL, на который транслируется код на C#, предназначенный для выполнения в среде .NET. Директива `.ctor` — это инструкция языка MSIL, предназначенная для создания конструктора класса. Не требуется исчерпывающее понимание языка MSIL, для того чтобы ознакомиться с содержимым модулей; мы просто будем обращать внимание на некоторые любопытные его особенности, по мере того как они будут встречаться.

В самом верху находится строка со слово MANIFEST; давайте теперь немного поговорим о манифестах.

Манифесты

В манифесте описываются все файлы или отсеки, из которых состоит данный модуль (модуль может состоять из нескольких файлов, хотя обычно он состоит только из одного файла). Что представляется еще более важным, в нем также описываются те внешние модули, на которые данный модуль ссылается. Например, если в программе используется `System.Data.dll`, то этот факт находит отражение в манифесте. Это значительно облегчает процедуру отслеживания зависимостей одних модулей от других, что, в свою очередь, существенно упрощает установку и проверку корректности инсталляции программы. Манифест также позволяет отслеживать номер версии модуля, что облегчает обновление программы. Рассмотрим манифест только что созданного модуля.

Щелкните два раза мышью на строке MANIFEST, расположенной в самой верхней части листинга, выведенного `ildasm` для `shapes.dll`. Это приведет к открытию нового окна, в котором будут содержаться подробности, относящиеся к манифесту (см. рис. справа).



В манифесте для `shapes.dll` содержатся две директивы `.assembly` и одна директива `.module`. Пока не следует беспокоиться по поводу содержимого блоков `.assembly`, т. е. временно можно не обращать внимания на то, что содержится внутри фигурных скобок. Самая первая строка выглядит следующим образом:

```
.assembly extern mscorlib
```

Эта строка представляет собой внешнюю ссылку на модуль `mscorlib.dll`, которая располагается в том же месте, где описывается большинство основополагающих классов `System` в .NET Framework. Такая внешняя ссылка является необходимой для каждой программы на C# и в .NET, если в ней используются классы `System`. Строка `.assembly Shapes` является объявлением самого модуля `shapes`. За ней следует объявление файла `shapes.dll` — строка `.module`. Наш модуль состоит всего лишь из одного файла, поэтому в нем содержится всего одно объявление `.module`.

Посмотрим, что произойдет, если добавить в исходный код дополнительную ссылку. Предположим, что требуется нарисовать некую геометрическую фигуру с применением одного из методов, описанных в пространстве имен `System.Drawing`. Закройте окно `ildasm` и вернитесь в проект Visual Studio.NET `shapes`. Внесите следующие изменения в файл `shapes.cs`. Во-первых, добавьте директиву `using`, ссылающуюся на пространство имен `System.Drawing`:

```
namespace Shapes
{
   using System.Drawing;

   public class Circle
   {
```

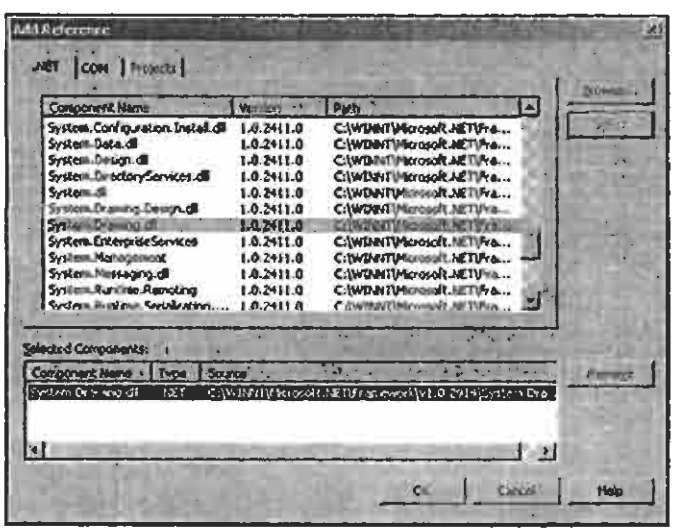
Затем добавьте метод `Draw()` для класса `Circle` сразу после метода `Area()`:

```
      public void Draw()
      {
         Pen p = new Pen(Color.Red);
      }
```

Этого кода совершено недостаточно, для того чтобы что-нибудь нарисовать на самом деле, но теперь вы понимаете к чему мы клоним!

Теперь требуется включить в проект ссылку на модуль `System.Drawing.dll`. Если этого не сделать, то мы увидим следующее сообщение об ошибке:

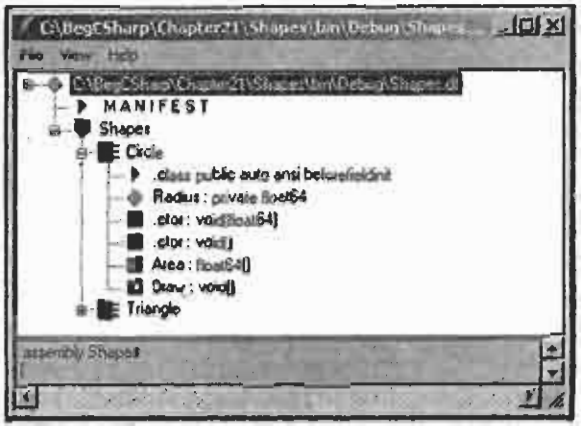
```
error CS0234: The type or namespace name 'Drawing' does not exist in the class or namespace
(ошибка CS0234: Название типа или пространства имен 'Drawing' отсутствует в классе или )
'System' (are you missing an assembly reference?)
(в пространстве имен 'System' (возможно, вы забыли указать ссылку на модуль? ))
```



Для добавления ссылки в VS, выберите пункт меню Project | Add Reference... . Внешний вид диалога Add Reference изображен на рисунке слева. Вкладка .NET позволяет просмотреть все доступные системные модули .NET; прокрутите список вниз и выберите модуль `System.Drawing.dll`. Затем щелкните мышью на кнопке Select.

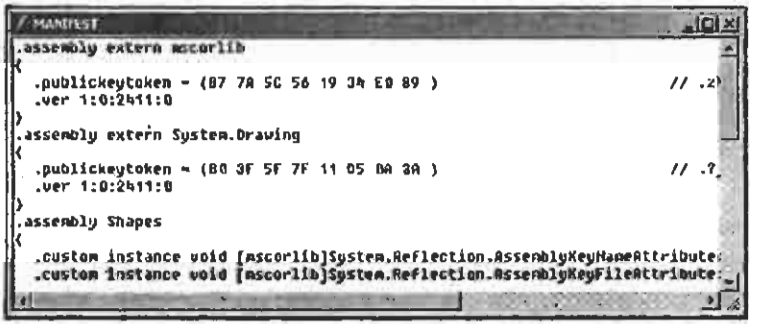
Чтобы добавить ссылку, следует нажать кнопку OK. В результате этого ссылка на `System.Drawing` будет включена в файл `shapes.dll`. Сохраните изменения, внесенные в исходный файл, закройте `ildasm` (в противном случае в процессе компиляции возникнет ошибка, поскольку `ildasm` использует открытый файл `shapes.dll`) и откомпилируйте файл `shapes.dll` заново с учетом внесенных в него изменений.

После этого повторно запустите `ildasm` и откройте файл `shapes.dll`. Теперь вы обнаружите метод `Draw()` внутри объекта `Circle`, как показано на рисунке справа.



Для того чтобы увидеть изменения, внесенные в MANIFEST, снова два раза щелкните мышью на указанной строке. Теперь в манифесте содержится ссылка на внешний модуль `System.Drawing`, что можно видеть на следующем ниже рисунке.

Модуль `shapes` информирует систему о том, что модуль `System.Drawing` требуется каждый раз, когда происходит обращение к самому модулю `shapes`. Скоро мы перейдем к рассмотрению того,



каким образом другая программа сможет воспользоваться модулем `shapes`, хотя взглянув на экран, изображенный на рисунке слева, вы, вероятно, на самом деле начинаете недоумевать по поводу того загадочного содержимого директив `.assembly`, которое мы порекомендовали временно игнорировать.

Атрибуты модулей

Помимо информации о ссылках на внешние модули в манифесте каждого модуля содержится и другая информация, которая относится непосредственно к самому модулю. Эта информация называется **атрибутами модуля**.

Файл AssemblyInfo.cs

Если вы создаете проект библиотеки классов в Visual Studio.NET, то вы, вероятно, обратили внимание на то, что в качестве составной части проекта создается второй исходный файл на C#. Этот файл называется `AssemblyInfo.cs` и используется для задания свойств модуля, хранящихся в манифесте, таких, как номер версии модуля, имя и т. д. Рассмотрим содержимое этого файла (для экономии места из него будут удалены некоторые комментарии):

```
using System.Reflection;
using System.Runtime.CompilerServices;

//
// Общая информация о модуле контролируется посредством
// следующего набора атрибутов. Для того чтобы внести
// изменения в информацию, относящуюся к данному модулю,
// необходимо изменить значения этих атрибутов.
//
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
```

```
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

//
// Информация о версии данного модуля состоит из
// следующих четырех значений:
//
//      Major Version          (старший номер версии)
//      Minor Version          (младший номер версии)
//      Build Number           (номер построения)
//      Revision               (исправление)
//
// У вас есть возможность либо самостоятельно определить
// все значения, либо использовать для атрибутов Revision
// и Build Numbers значения по умолчанию, для чего следует
// использовать '*', как показано ниже:

[assembly: AssemblyVersion("1.0.*")]

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

Каждый из операторов, заключенных в квадратные скобки и имеющих вид `[assembly: AssemblyXxxxx]`, представляет собой *атрибут*, т. е. специальный элемент C#, который более подробно рассматривается в следующей главе. В данный момент вполне достаточно знать, что каждый из этих операторов задает значение какого-либо конкретного свойства модуля. Слово `assembly:`, расположенное в начале каждого атрибута, сообщает системе, что следующая далее директива имеет отношение к самому модулю, а не к классу, методу или какой-либо другой части программы.

VS создает этот файл в качестве шаблона, который может быть заполнен в соответствии со свойствами, которые вы хотите придать модулю. Можно использовать эти атрибуты в исходном файле, откомпилированном с помощью директивы `csc`. Большая часть атрибутов, среди которых название, компания и т. п., несут исключительно информационную нагрузку, и им может быть присвоено любое описательное значение, которое вы захотите поставить в соответствие модулю, например:

```
[assembly: AssemblyTitle("MyCompany Shapes Class Library")]
[assembly: AssemblyDescription("Classes for Manipulation of Shapes")]
[assembly: AssemblyConfiguration("Enterprise Version")]
[assembly: AssemblyCompany("MyCompany, Inc.")]
[assembly: AssemblyProduct("Shapes")]
[assembly: AssemblyCopyright("Copyright 2001, MyCompany, Inc.")]
[assembly: AssemblyTrademark("Shapes is a trademark of MyCompany, Inc.")]
```

Атрибут `AssemblyCulture` касается национального языка, который используется для данного модуля, и если таковой задан, то он должен следовать специальным сокращениям в соответствии с международным стандартом. Более подробную информацию можно найти в пространстве имен `System.Globalization` и в темах, посвященных культуре онлайновой документации .NET Framework. Не требуется задавать этот атрибут до тех пор, пока вы не распространяете версии вашего компонента, работающие на различных языках; если же этот атрибут задан, то система выполнения программ .NET начинает автоматически искать версию модуля, которая соответствует текущему значению, установленному в системе. Например, во Франции будут выводиться сообщения на французском языке (для чего будут

использоваться ресурсы сообщений и/или код на французском языке), а в Великобритании сообщения будут выводиться на английском языке (будут использоваться ресурсы сообщений и/или код на английском языке). Для того чтобы все происходило именно таким образом, необходимо пометить конкретный модуль с помощью соответственного значения атрибута.

Раз эти атрибуты появляются в файле `AssemblyInfo.cs`, они будут также включены в манифест модуля. Если теперь снова взглянуть на манифест `shapes.dll`, то можно обнаружить следующую строку в разделе `.assembly Shapes`:

```
.custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::
.ctor(string) = ( 01 00 00 00 00 )
```

Эта строка указывает на то, что в модуле содержится атрибут `AssemblyTitle`. Если файл `shapes.cs` будет откомпилирован из командной строки без использования файла `AssemblyInfo.cs`, то такая строка в манифесте будет отсутствовать.

Номера версий

Номер версии в .NET Framework состоит из четырех частей, как показано ниже:

Major version (старший номер версии)	Minor version (младший номер версии)	Build Number (номер построения)	Revision (исправление)

Первые две части номера, скорее всего, знакомы вам как пользователю потребительского программного обеспечения — это старший номер версии и младший номер версии, как например в `Shapes` версии 1.0 (где 1 — старший номер версии, а 0 — младший).

Следующие две части номера версии позволяют еще больше его детализировать. Номер построения обозначает число раз, которое строился данный модуль; этот номер обычно изменяется каждый раз, когда модуль строится заново.

Номер исправления дает еще один уровень детализации и предназначается для нумерации "заплаток" или исправленных ошибок в данном модуле, который совершенно ничем не отличается от своего предшественника, кроме как этой единственной исправленной ошибкой.

Атрибуты версий

В `ildasm` можно просмотреть атрибуты версии модулей. Взгляните на манифест файла `shapes.dll`, и вы увидите, что для всех модулей, на который ссылается названный манифест, внутри каждого блока `.assembly` существует директива `.ver` для данного модуля.

Информация о номере версии внешних модулей, на которые ссылается данный модуль, например, на модули `System.Xxxx` и на CLR (`mscorlib`), имеет одинаковый вид. Это происходит потому, что система выполнения программ .NET использует номера версий, хранящиеся в модулях, для проверки совместимости. Мы подробнее остановимся на этом вопросе, когда перейдем к обсуждению проблемы совместимости версий.

Если вами использовался компилятор командной строки (`csc`), то в строке с директивой `.ver` для модуля `shapes` будут содержаться одни нули:

```
.ver 0:0:0:0
```

Это происходит потому, что до сих пор модулю shapes не присвоен никакой номер версии. Если же вы используете Visual Studio.NET, то вы, вероятно, сможете увидеть приблизительно следующий номер версии:

```
.ver 1:0:486:7484
```

благодаря тому, что VS включает информацию о номере версии автоматически.

Атрибут AssemblyVersion

Внутри файла AssemblyInfo.cs, создаваемого VS, номер версии задается посредством атрибута AssemblyVersion:

```
[assembly: AssemblyVersion("1.0.*")]
```

Атрибут AssemblyVersion допускает использование символа звездочки (*) для последних двух частей номера версии. Это означает требование того, чтобы VS автоматически устанавливал номера построения и исправления. Существует также возможность использовать звездочку только для номера исправления (как в 1.1.1.*), но не для старшего и младшего номеров версии (вариант 1.* является недопустимым). Если вы посмотрите на номер версии модуля с помощью ildasm, то увидите, что текущий номер версии, установленный VS, имеет приблизительно следующий вид:

```
.ver 1:0:585:24784
```

Если внести какие-либо изменения в код классов и построить проект заново, то можно увидеть, что номер версии изменился автоматически и принял приблизительно следующий вид:

```
.ver 1:0:585:25005
```

Можно также непосредственно задать все части номера версии, указав какой-либо конкретный номер вместо звездочки:

```
[assembly: AssemblyVersion("1.0.1.2")]
```

Это приведет к созданию VS модуля с данным конкретным номером версии.

Если вы разрабатываете программу для своего собственного использования, то вам не следует устанавливать номер версии. Однако если вы собираетесь производить программное обеспечение для других конечных пользователей, то вам потребуется изменять старший и младший номера версии по мере того, как вы будете создавать новые варианты программы с добавлением новых существенных функциональных возможностей: 1.0 для первого выпуска продукта, 1.1 для варианта, в который внесены менее значительные изменения, 2.0 для более существенных изменений и т. д. При этом допустимым является возложить обязанность на VS автоматически определять номера построения/исправления, и в большинство случаев вам даже не потребуется знать их значения.

Было бы очень удобно использовать ildasm для проверки полного номера версии. Например, если какой-либо конечный пользователь сообщает об обнаруженной ошибке, то у вас будет возможность сравнить номер версии модуля на своем компьютере с версией, установленной на компьютере этого пользователя. Вы сможете совершенно точно определить номер версии, с которой работает пользователь; используя номера построения и исправления можно даже отличить построение, которое вы осуществили сегодня утром, от сделанного вами сегодня перед обедом.

Совместимость версий

Система выполнения программ .NET осуществляет проверку номеров версий при загрузке модулей с целью определения совместимости версий. Такая проверка осуществляется только для **модулей общего доступа**, к рассмотрению которых мы приступим в настоящей главе позднее. Однако способ, посредством которого программа проверки номеров версий выполняет свою работу, можно обсудить прямо сейчас.

В манифесте модуля содержатся номер версии данного модуля и номера версий внешних модулей, на которые имеются ссылки. Когда система выполнения программ .NET осуществляет загрузку модуля, на который имеется ссылка, она производит сличение номера версии, содержащегося в манифесте загружаемого модуля, с номером версии, указанном в ссылке на этот модуль, для того чтобы убедиться в совместимости версий.

Если у модулей оказываются различными старшие или младшие номера версий, они считаются несовместимыми, а модуль, на который указывает ссылка, не будет загружен — например, программа Shapes версии 1.1 будет несовместима с программой, которая ссылается на Shapes версий 1.0, 1.2 или 2.0.

Но как быть, если на одном и том же компьютере используются программа A, работающая с версией Shapes 1.0, и программа B, работающая с версией Shapes 1.1? Этим вопросом как раз и занимается система выполнения программ .NET.; такая особенность называется параллельным выполнением (side-by-side execution) и обеспечивает возможность инсталляции на одном и том же компьютере обеих версий программы Shapes — 1.0 и 1.1, каждая из которых оказывается доступной любой ссылающейся на них программе.

Следующие две части номера версии выводят работу с номерами версий на более тонкий уровень детализации. Номер построения определяет, каким по счету является построение данной программы; этот номер обычно претерпевает изменения каждый раз, когда модуль строится заново. Два модуля, обладающих одинаковыми старшими и младшими номерами версии и различными номерами построений, могут быть (а могут и не быть) совместимыми; система выполнения программ предполагает, что они являются совместимыми, и допускает их загрузку; задача вносить изменения в старший и младший номера версии в том случае, если программа претерпевает изменения, делающие ее несовместимой с предыдущими версиями, возлагается на разработчика.

Оба номера версий модулей `mscorlib` и `System.Drawing` соответствовали версии 1.0 с номером построения 2411 — это номер построения системных модулей .NET, которые используются на моем компьютере в тот момент, когда я пишу этот текст. Если какая-либо из этих библиотек будет заменена на новую с номером построения 2412 или даже 9999, то система выполнения программ .NET будет пытаться использовать вновь построенную версию, хотя никаких гарантий ее совместимости не существует.

Номер исправления представляет собой еще один уровень детализации, позволяя определять некое конкретное изменение или исправление ошибки для данного номера построения. Предполагается, что версии 1.0.2411.0 и 1.0.2411.1 являются полностью совместимыми. Для сценария, при котором каждая конкретная программа выполняется с другой конкретной программой, если два одновременно существующих модуля обладают совпадающими старшим и младшим номерами версии и отличаются только номером построения и/или номером исправления, то система будет пытаться выполнить самый новый модуль (другими словами тот, у которого номер построения/исправления окажется больше).

Вызов модулей

Перейдем к рассмотрению вопроса о том, что происходит, если в какой-либо программе появляется ссылка на shapes. Мы создадим простого клиента, использующего shapes, и назовем его shapeuser.cs:

```
namespace ShapeUser
{
   using System;
   using Shapes;

   public class ShapeUser
   {
      public static void Main()
      {
         Circle c = new Circle(1.0F);
         Console.WriteLine("Area of Circle(1.0) is {0}", c.Area());
      }
   }
}
```

Компиляция файла shapeuser с помощью командной строки

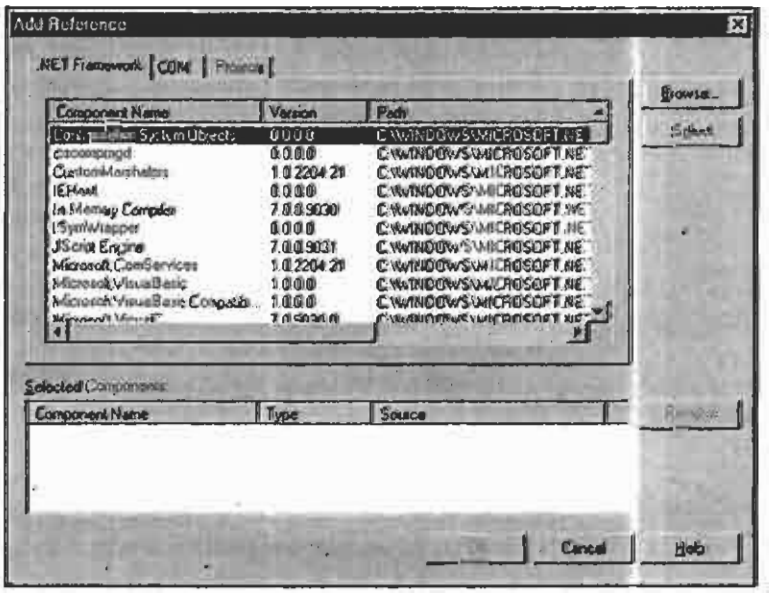
Если вы используете компилятор C# из командной строки, то следует создать файл shapeuser.cs в той же самой директории, в которой располагаются файлы shapes.cs и shapes.dll, и набрать следующую команду:

```
csc /reference: shapes.dll shapeuser.cs
```

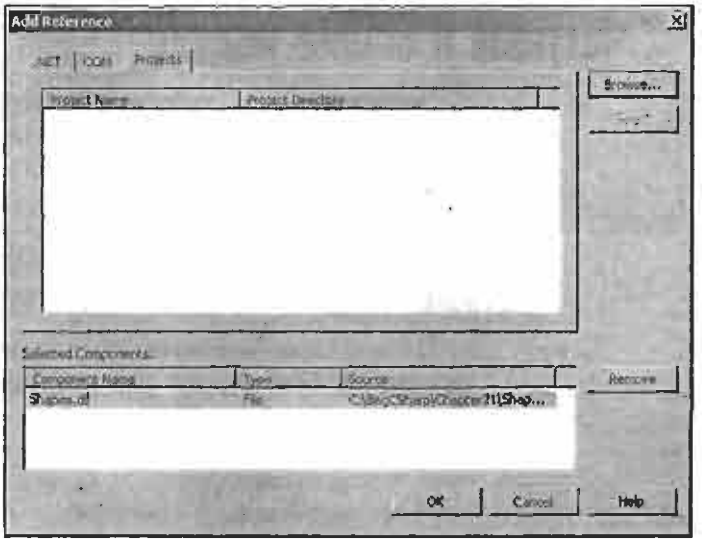
Это приведет к созданию в текущей директории файла shapeuser.exe, который будет ссылаться на ddl.shapes.

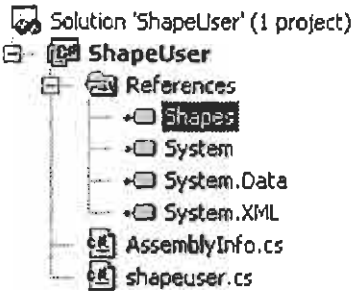
Компиляция файла shapeuser с помощью VS

Если вами используется VS, то создайте новый проект консольного приложения с именем ShapeUser. Переименуйте исходный файл под именем Class1.cs в shapeuser.cs введите в него программу, приведенную выше. Этому проекту при компиляции потребуется ссылка на проект Shapes, поэтому вам следует выбрать пункт Add Reference из меню Project (или щелкните правой кнопкой мыши на References в Solution Explorer и выберите Add Reference...), в результате чего будет открыт диалог Add Reference, представленный на рисунке справа.



Щелкните мышью на вкладке Projects, а затем нажмите кнопку Browse (расположенную справа вверху). В диалоговом окне Select Components (см. рис. справа) найдите файл `shapes.dll` в директории вашего проекта `Shapes` и щелкните мышью кнопку Open. Файл `shapes.dll` будет включен в список выбранных компонентов Selected Components.



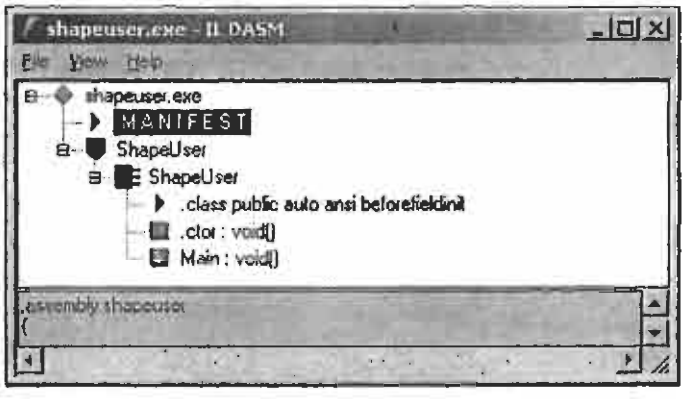


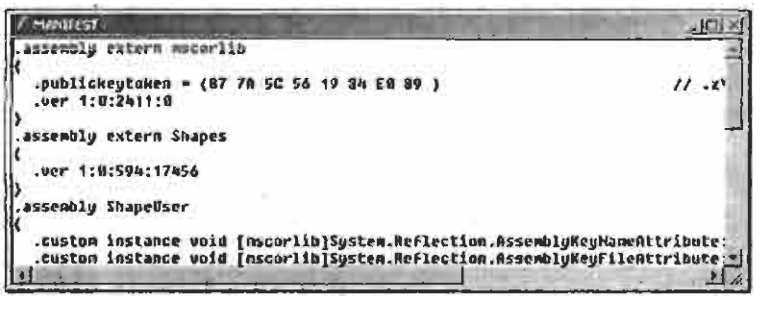
Нажмите кнопку OK — это приведет к тому, что `shapes` будет добавлен к списку ссылок проекта `ShapeUser` (см. рис. слева).

В результате в директорию `bin\Debug` проекта `ShapeUser` будет добавлена частная копия `shapes.dll`. Подробнее обсудим это в следующем разделе, а пока осуществим построение приложения `ShapeUser`.

Исследование проекта с помощью ILDASM

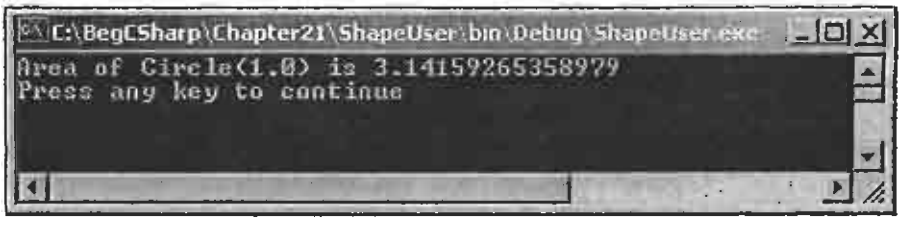
Теперь, когда `ShapeUser.exe` построен, можно использовать `ildasm` для исследования его содержимого (см. рис. справа).





Обратите внимание на то, что, хотя это и автономная программа, она обладает такими же метаданными, как и у библиотеки классов. Щелкните два раза мышью на MANIFEST, и вы увидите в окне, представленном на рисунке слева, внешнюю ссылку на компонент `shapes`.

Теперь можно выполнить `shapeuser.exe`, в результате чего увидим то, что изображено на рисунке справа.



Частные и общие модули

До настоящего момента нам приходилось иметь дело только с **частными модулями** — т. е. модулями, устанавливаемыми в качестве составной части единственного приложения. В .NET имеются специальные средства, позволяющие создавать **общие модули**, которые могут использоваться несколькими программами одновременно.

Частные модули

По умолчанию модуль является частным по отношению к проекту. Частные модули обязательно должны располагаться в той же директории, что и само приложение.

Модуль `shapes.dll` является частным; для того чтобы ссылаться на него из проекта `Shapes`, нам пришлось на предыдущем этапе строить оба проекта в одной и той же директории, в противном случае потребовалось бы осуществлять поиск по директории разработки проекта `Shapes` из проекта `ShapeUser` при добавлении ссылки в VS; в этом случае VS создает частную копию `shapes.dll` и размещает ее в директории `ShapeUser`.

Создание копии `shapes.dll` позволяет гарантировать, что приложение `ShapeUser` будет работать, даже если оригинальный файл `shapes.dll` окажется недоступным из-за продолжающегося процесса разработки. Однако подход с созданием копий каждой динамически загружаемой библиотеки для широко используемых компонентов оказывается не слишком эффективным, поэтому в .NET Framework предусмотрено использование общих модулей.

Общие модули

Общие модули оказываются доступными для использования всеми программами, выполняющимися на данной системе. От программы не требуется, чтобы она располагала информацией о местоположении общего модуля, поскольку все общие модули хранятся в специальной системной директории .NET, которая называется **Global Assembly Cache** (GAC — **кэш глобальных модулей**). Поскольку эти модули доступны по всей системе, то на систему выполнения программ .NET налагается обязанность выполнения нескольких дополнительных проверок, касающихся общих модулей и позволяющих определить допустимость их использования программой, которая к ним обращается, например, по вопросам обеспечения безопасности и совместимости версий.

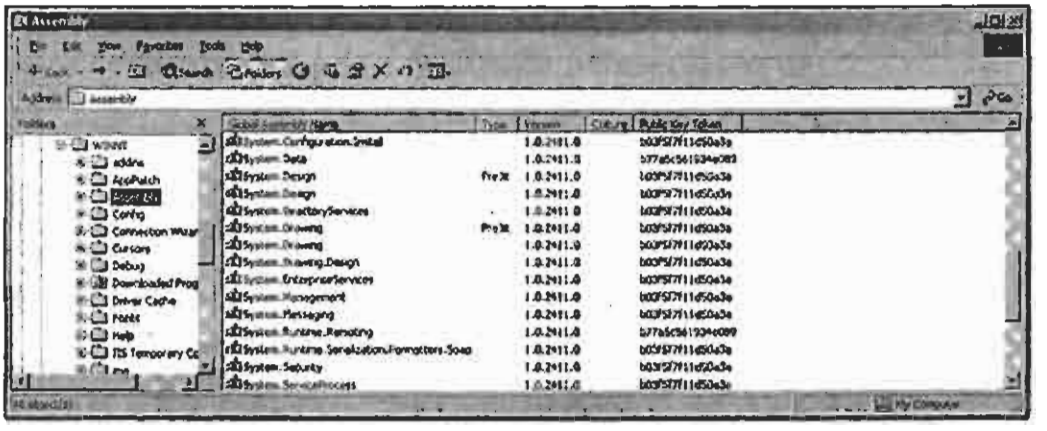
Безопасность и строгие имена

Любой общий модуль должен обеспечивать гарантию того, что он не был заменен на какой-либо другой модуль, обладающий теми же именем и номером версии, но в который были внесены те или иные изменения, например, вследствие заражения вирусом. Это достигается за счет того, что для общего модуля существует дополнительное требование — он должен быть **помечен** зашифрованным ключом, прежде чем он будет загружен в глобальный кэш модулей. Такой ключ обеспечивает защиту не только от нарушения системы обеспечения безопасности, но и от обычных коллизий, связанных с тем, что у двух компонентов оказываются одинаковые имя и номер версии. Если значения ключей у компонентов оказываются различными, то такие компоненты не считаются одинаковыми, даже несмотря на то, что они обладают идентичными именами. Уникальное сочетание, состоящее из имени модуля, номера его версии и его ключа, называется **строгим именем**.

Кэш глобальных модулей

Кэш глобальных модулей представляет собой специальную директорию, которая располагается в директории `WinNT\assembly`. Все общие создаваемые компанией Microsoft модули, включая модули `System` .NET Framework, хранятся в этой директории и загружаются именно оттуда. Если вы попытаетесь просмотреть эту директорию с помощью Windows Explorer, то вы увидите специальное внешнее расширение Windows, в котором выводятся различные свойства модулей, в том числе и ключ, входящий в состав строгого имени каждого из них.

Внешнее расширение Windows (называемое `shfusion.dll`) встроено в Windows Explorer и призвано расширить его возможности по сравнению с обычным выводом файлов. На следующем рисунке изображен тот вид, который GAC имеет в Windows Explorer. При этом осуществляется вывод имени, номера версии, ключа и других свойств модулей:



Внешнее расширение Windows делает обязательным применение политики обеспечения безопасности для глобальных модулей, позволяя копировать модули в эту директорию посредством простого переноса мышью только в том случае, если соблюдены все необходимые правила.

> *Чтобы иметь доступ к кэшу глобальных модулей, необходимо обладать правами локального администратора.*

Создание общего модуля

Чтобы создать общий модуль со строгим именем, требуется сгенерировать два ключа — общий и частный, которыми будет помечен этот модуль. Асимметричные системы шифрования на основе общих/частных ключей используют частный ключ, *известный* только отправителю закодированного сообщения, и общий ключ, который открыт для всех. В среде .NET используется такой же механизм, чтобы гарантировать, что общий модуль, на который делается ссылка,— это действительно тот модуль, который требуется (модули, создаваемые различными компаниями, могут обладать одинаковыми именами и номерами версий, а то какой-нибудь хакер попытается "вскрыть" программу, создав модуль с теми же именем и номером версии или испортив содержимое существующего модуля). Ключи, содержащиеся в вызове модуля и непосредственно в (помеченном) модуле, сравниваются, и если они не совпадают, то загрузка общего модуля не производится.

Сочетание имени модуля, номера его версии и общего ключа гарантированно является уникальным; такое сочетание называется **строгим именем**.

В .NET Framework предусмотрен специальный инструмент `sn.exe`, предназначенный для создания строгих имен (`sn` расшифровывается как Strong Name — строгое имя). К сожалению, этот инструмент может использоваться только из командной строки, и придется настроить переменную среды PATH перед тем, как вы сможете воспользоваться им (инструкции содержатся в приложении А).

Запустите командную строку с помощью VS Command Prompt (выбрав пункт меню View | Other Windows | Command Window) или откройте окно командной строки непосредственно в Windows. Для того чтобы создать ключ, потребуется задать некоторое имя файла (обычно с расширением `.snk`), к которому вы собираетесь обращаться из своего модуля:

```
sn -k shapes.snk
```

Это приведет к созданию файла с именем `shapes.snk` в текущей директории. Для того, чтобы пометить модуль с помощью данного ключа, модифицируйте атрибут `AssemblyKeyFile`, находящийся в последней части файла `AssemblyInfo.cs` вашего проекта:

```
[assembly: AssemblyKeyFile("shapes.snk")]
```

В VS местонахождение ключа является относительным к директории вашего проекта `obj\debug` или `obj\release`; если вы помещаете файл в какую-либо другую директорию, например, в корневую директорию проекта, то необходимо задать путь к этой директории: `[assembly: AssemblyKeyFile(@"..\..\shapes.snk")]`.

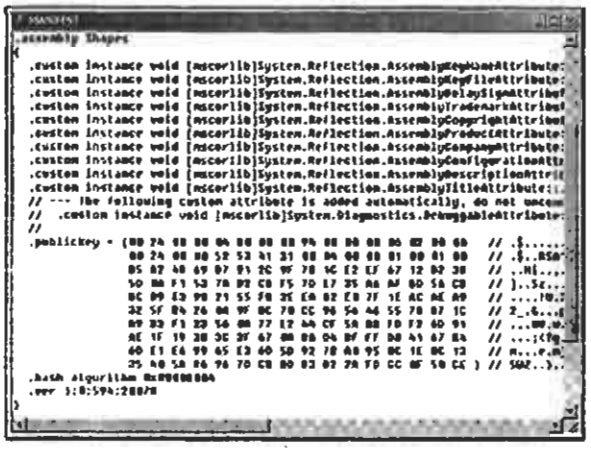
> *Обратите внимание на то, что при использовании компилятора командной строки следует просто добавить строку `AssemblyKeyFile` в начало вашего файла `shapes.cs`, перед которой должны располагаться соответствующие директивы `using`.*

```
using System.Reflection;
using System.Runtime.CompilerServices;
[assembly: AssemblyKeyFile("shapes.snk")]
```

Теперь необходимо заново откомпилировать модуль `shapes.dll`. Если VS оказывается не в состоянии обнаружить файл `shapes.snk`, то проект не компилируется и выдается сообщение об ошибке следующего вида:

```
Cryptographic failure while signing assembly
(Ошибка шифрования при попытке пометить файл)
'C:\BegCSharp\Chapter21\Shapes\obj\Debug\Shapes.dll' — 'Error reading key file 'shapes.snk'
('C:\BegCSharp\Chapter21\Shapes\obj\Debug\Shapes.dll' — 'Ошибка при попытке
прочитать файл 'shapes.snk')
— The system cannot find the file specified.
(— Система не в состоянии обнаружить указанный файл.)
```

В этом сообщении содержится точная информация относительно того, где именно VS осуществляет поиск файла с ключом, и вы можете либо перенести туда требуемый файл, либо указать путь к директории, в которой он на самом деле расположен.



После того как проект будет успешно построен заново, модуль будет считаться помеченным. Если теперь внимательно посмотреть на манифест модуля `shapes.dll` с помощью `ildasm`, то можно обнаружнть вновь созданный общий ключ и встроенный внутрь модуля (см. рис. слева).

Обратите внимание на то, что если вы прямо сейчас попытаетесь запустить `shapeuser.exe`, не откомпилировав его заиово, то вы получите приведенное ниже сообщение об ошибке, поскольку модуль `shapes.dll` больше не соответствует обращению к нему (внешняя ссылка указывает на частный модуль, а `shapes` теперь является общнм модулем):

Unhandled Exception: System.IO.FileLoadException: The located assembly's manifest definition with name 'shapes' does not match the assembly reference. File name: "shapes"
(Необрабатываемая исключительная ситуация: System.IO.FileLoadException: Имеющееся определение) (манифеста модуля с именем 'shapes' не соответствует ссылке на этот модуль. Имя файла: "shapes")

at ShapeUser.ShapeUser.Main()

Этой ошибки не произошло бы, если бы мы создали `ShapeUser` с помощью VS, поскольку VS создает частную копию модуля, а эта копия не претерпевает никаких изменений.

Теперь необходимо повторио откомпилировать файл `shapeuser.cs`, что позволит обновить ссылку на внешний модуль, находящуюся в `shapeuser.exe` в соответствии с помеченной версией `shapes.dll`. После повторной компиляции все будет работать как прежде, поскольку при этом используется локальная копия `shapes.dll`.

В завершенне необходимо скопнровать модуль, который теперь обладает строгим именем, в кэш глобальных модулей (`WinNT\assembly`). В качестве альтернативы можно воспользоваться специальным инструментом командной строки .NET, который иазывается `gacutil` (Global Assembly Cache Utility — утилита для работы с кэшем глобальных модулей). Чтобы инсталлировать `shapes.dll` в этот глобальный кэш, следует воспользоваться ниструментом `gacutil` с флагом `/i` из командной строки Visual Studio.NET:

```
gacutil /i shapes.dll
```

Полученное сообщение "Assembly successfully added to the cache" свидетельствует об успешной инсталляции данного модуля.

Прн установке коммерческого приложення наиболее предпочтительным способом включения общего модуля в глобальный кэш является использование Windows Installer.

Чтобы убедиться в том, что `shapes.dll` занесен в кэш, попробуйте удалить копию `shapes.dll` из текущей директории. Запустите файл `shapeuser.exe`, и вы получите следующее сообщение:

```
Area of Circle(1.0) is 3.141596
```

Программа по-прежнему работает, несмотря на отсутствие `shapes.dll`, поскольку загрузка модуля `shapes` осуществляется из GAC. Для продолжения тестирования воспользуемся `gacutil` для обратной инсталляции `shapes`:

```
gacutil /using shapes
```

Обратите внимание на то, что для команды обратной инсталляции расширение `.dll` опускается.

Теперь попробуйте еще раз запустить `shapeuser`:

```
C:\shapes>shapeuser
Unhandled Exception: System.IO.FileNotFoundException: File or assembly name shapes, or one of
(Необрабатываемая исключительная ситуация: System.IO. FileNotFoundException:
Имя файла или модуля shapes, либо файла)
its dependencies, was not found.
(или модуля, на которые они ссылаются, обнаружить не удалось.)
```

Это свидетельствует о том, что `shapes.dll` был предварительно загружен из GAC.

Поиск модулей

Система выполнения программ .NET следует набору предварительно описанных правил для определения местонахождения внешнего модуля, если на него имеется ссылка. Мы только что познакомились с тем, каким образом осуществляется поиск общих модулей сначала в локальной директории, а затем в GAC.

Для частных модулей поиск сначала выполняется в локальной директории, а затем система осуществляет поиск в поддиректории с тем же именем, что и сам модуль. Система выполнения программ также осуществляет поиск файла DLL или EXE, который носит то же имя, что и искомый модуль. Для класса `Shapes` сочетание этих правил дает следующий перечень просматриваемых директорий:

```
./shapes.DLL
./shapes/shapes.dll
./shapes/shapes.exe
./shapes.exe
```

Дополнительные области поиска и даже адреса URL, по которым данный модуль может быть найден и загружен с удаленного местонахождения, могут задаваться посредством **файла конфигурации**. Файлы конфигурации для модулей представляют собой файлы в формате XML, определяющие правила, которые система выполнения программ .NET должна применять при поиске модуля. Файлы конфигурации также позволяют предопределять поведение, реализующееся при проверке номеров версий. Детальное описание файлов конфигурации и XML оказывается достаточно сложным и выходит за рамки настоящей книги. Для получения такой информации можно обратиться к документации по .NET Framework, доступной в режиме онлайн, по теме "файлы конфигурации модуля".

Итоги

Программы на C# и библиотеки классов распространяются в виде модулей, которые обладают большим количеством возможностей, облегчающих распространение компонентов Microsoft .NET Framework. Компоненты предоставляют возможность повторного использования двоичных объектов. Очень важной особенностью .NET-компонентов является их самоописательность; это их самое главное их отличие от исторических предшественников (например, от COM-компонентов).

Свойство самоописательности обладает определенными преимуществами, среди которых простота инсталляции и совместного использования с Common Language Runtime (CLR — единая система выполнения программ), что позволяет одновременно с программированием на C# использовать различные другие языки программирования и унаследованные программы.

Нами был создан компонент, представляющий собой написанную на C# библиотеку классов, который был затем откомпилирован в модуль с использованием как инструментария командной строки .NET, так и Visual Studio.NET. После этого мы создали приложение на C#, в котором данный компонент использовался, а также познакомились с тем, каким образом можно просматривать содержимое модулей с помощью `ildasm`. Это позволило нам лучше понять структуру модулей.

Мы рассмотрели различные составные части модуля, включая манифест, номер версии и другие атрибуты модуля. Кроме лучшего понимания структуры модулей, это позволило нам научиться выполнять сравнение номеров версий в ссылках на внешние модули с целью отладки программ.

Модули могут быть либо частными, относящимся только к некоторому конкретному приложению, либо общими, которые доступны всем приложениям, работающим в данной системе. Мы познакомились с тем, каким образом можно генерировать общие модули, создавая ключ файла и помечая этим ключом данный модуль, в результате чего создается строгое имя модуля. Мы познакомились также с тем, каким образом осуществляется проверка номеров версий для общих модулей, и рассмотрели понятие кэша глобальных модулей, в котором хранятся общие модули, а также различные инструменты, предназначенные для просмотра и выполнения манипуляций над ними.

Мы рассмотрели компромисс между преимуществами проверки совместимости, обеспечения безопасности и эффективности использования ресурсов с одной стороны, и теми трудностями, к которым это приводит при создании и инсталляции модулей, с другой. В заключение мы коротко рассмотрели, каким образом система выполнения программ .NET осуществляет поиск модулей, к которым происходит обращение.

Все эти возможности модулей существенно упрощают разработку, выполняемую в среде .NET.

Упражнения

- Измените номер версии и другие атрибуты модуля в `AssemblyInfo.cs` или `shapes.cs`, и понаблюдайте с помощью `ildasm` за тем, к каким изменениям в файле `shapes.dll` это приведет.
- Создайте новый модуль `moreshapes`, в который — наряду с классами `Circle` и `Triangle` — будет включен класс `Square` (квадрат), а затем изучите его свойства с помощью `ildasm`.
- Замените в `shapeuser.exe` ссылку на модуль, с тем чтобы воспользоваться модулем `moreshapes.dll`, а затем посмотрите в `ildasm`, какие изменения произошли со свойствами.
- Создайте клиента для `moreshapes`, который будет использовать объекты `Square` и `Triangle` наряду с объектом `Circle`. С помощью `ildasm` исследуйте модуль, созданный для этого клиента.

- Выведите список командных опций `gacutil` с помощью флажка `/?`, а затем воспользуйтесь его опциями для вывода перечня свойств всех глобальных модулей, которые имеются на вашей системе.
- Создайте строгое имя для модуля `moreshapes` и пометьте его с помощью этого имени. Посмотрнте на результат с помощью `ildasm`. Попробуйте запустить клиентскую программу, а затем откомпилируйте ее заново, включив ссылку на помеченный модуль.
- Инсталлируйте `moreshapes.dll` в Global Assembly Cache с помощью `gacutil` и поэкспериментируйте с `shapeuser` и созданным вами клиентом, запуская их как совместно с модулем `moreshapes`, находящимся в локальной директории и/или в GAC, так и без него.

22

Атрибуты

В этой главе будет рассказано о том, что такое атрибуты и для чего они могут быть использованы, а в качестве примера будут приведены некоторые атрибуты, используемые в .NET Framework.

Будет также обсуждено понятие нестандартных атрибутов — атрибутов, которые вы можете создавать самостоятельно для расширения вашей системы, и приведено несколько работающих примеров. Будет рассмотрен вопрос о том, каким образом Intermediate Language Disassembler (`ildasm` — дисассемблер промежуточного языка) может использоваться для обнаружения атрибутов существующих модулей.

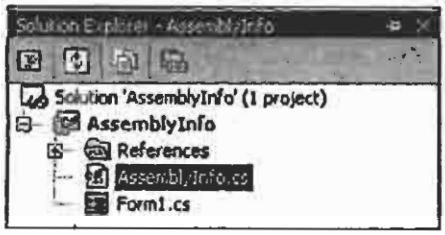
Атрибуты являются одной из наиболее полезных особенностей .NET Framework. Они часто используются компанией Microsoft. Для того чтобы научиться эффективно их использовать, придется потратить значительное количество времени, но оно того стоит. В последующих разделах мы познакомимся с тем, каким образом атрибуты могут применяться в следующих областях:

- Отладка
- Обеспечение информацией о модулях
- Обозначение методов и классов как устаревших
- Условная компиляция
- Доступ к базам данных

В этой главе будет подробно рассказано о том, как создавать собственные атрибуты, расширяющие возможности системы, и продемонстрировано несколько работающих примеров нестандартных атрибутов. Прочитав главу, вы получите достаточно знаний о том, что из себя представляют атрибуты и каким образом их можно использовать в своих проектах.

Что такое атрибут?

Дать атрибуту определение, состоящее из одного предложения, довольно сложно; изучать атрибуты лучше всего на примере их использования. На данном этапе мы определим атрибут как некоторую дополнительную информацию, которая может относится к отдельным участкам кода в рамках одного модуля, таким как класс, метод или свойство. Эта информация является доступной для любого другого класса, который обращается к данному модулю.



В предшествующей главе мы обсуждали понятие модулей и упоминали файл `AssemblyInfo.cs`. Для того чтобы рассмотреть этот файл, следует создать в VS новый проект и открыть Solution Explorer (*Ctrl+Alt+L*). Вы увидите нечто похожее на изображенное на рисунке слева.

Если два раза щелкнуть мышью на этом файле, можно увидеть код, который был создан VS. Часть этого кода приводится ниже:

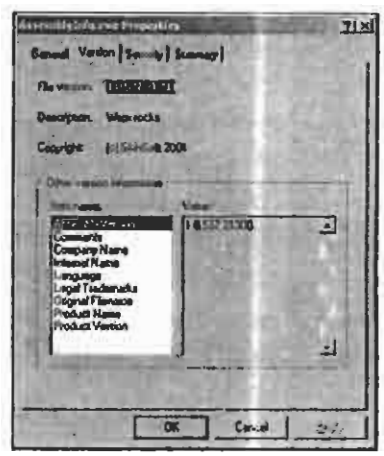
```
using System.Reflection;
using System.Runtime.CompilerServices;

//
// Общая информация о модуле контролируется посредством
// следующего набора атрибутов. Для того чтобы внести
// изменения в информацию, относящуюся к данному модулю,
// необходимо изменить значения этих атрибутов.
//
[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

Для краткости здесь приводится только часть файла. Внутри файла находится некоторое количество строк, начинающихся с текста `"[assembly:"`,— это и есть определения атрибутов. Когда такой файл компилируется, все описанные атрибуты сохраняются в получающемся модуле. Этот процесс известен под названием 'консервирования' ('pickling'). Чтобы увидеть это в действии, измените один из приведенных выше атрибутов (например, атрибут `AssemblyTitle`) и откомпилируйте модуль:

```
[assembly: AssemblyTitle("Wrox rocks")]
```

После завершения компиляция щелкните мышью на модуле в Windows Explorer, что приведет к выводу на экран окна со свойствами. На рисунке справа показана вкладка с информацией о версиях. В поле **Description** находится описание содержимого атрибута `AssemblyTitle`.



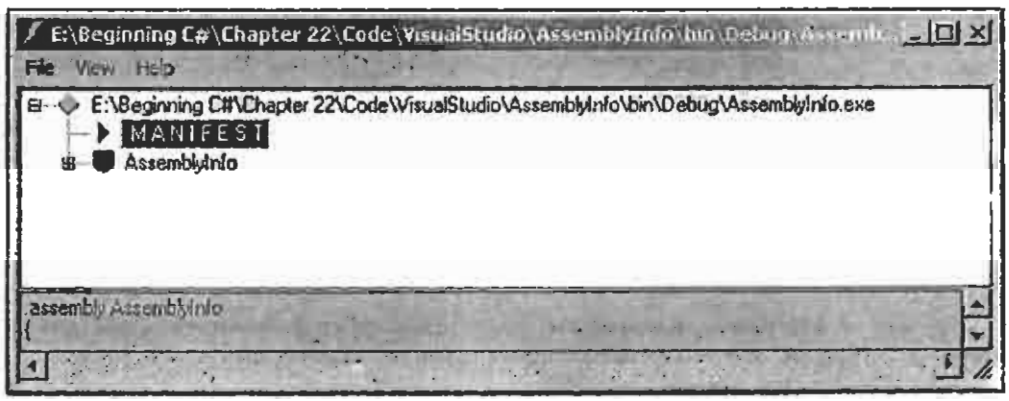
Атрибут	Информация о версии
`AssemblyTitle`	Description (описание)
`AssemblyDescription`	Comments (комментарии)
`AssemblyCompany`	Company Name (название компании)
`AssemblyTrademark`	Legal Trademarks (официальные торговые марки)
`AssemblyVersion`	Assembly Version and Product Version (версия модуля и версия продукта)
`AssemblyCopyright`	Copyright (копирайт)
`AssemblyProduct`	Product Name (наименование продукта)

В таблице слева перечислены атрибуты модуля и соответствующие им имена на вкладке с информацией о версии.

Перечень атрибутов, представленных в окне со свойствами, оказывается несколько меньше, чем перечень атрибутов, описанных в модуле. Компанией Microsoft предусмотрено отображение в окне свойств только некоторых, наиболее часто используемых атрибутов, а для того чтобы добраться до остальных атрибутов, требуется либо написать некоторый код (этот код приводится в следующем разделе), либо воспользоваться `ildasm`.

Для того чтобы найти все атрибуты, относящиеся к данному модулю, можно использовать `ildasm`, который позволяет просмотреть модуль и найти все описанные в нем атрибуты. Если вы до сих пор не включили `ildasm` в VS в качестве внешнего инструмента, то следует воспользоваться инструкциями о том, как это следует делать, приведенными в предыдущей главе.

Откройте `ildasm` и выберите модуль с помощью меню **File | Open**. После этого на экране можно увидеть следующее:



Щелкнув два раза мышью на выделенном разделе **MANIFEST**, вы откроете окно более низкого уровня, в котором находится манифест модуля, подробно описанный в предыдущей главе. Немного прокрутив это файл вниз, вы увидите несколько странно выглядящих строк кода:

```
.assembly AssemblyInfo
{
   .custom instance void
   [mscorlib]System.Reflection. AssemblyCopyrightAttribute::.ctor(string)
      = (01 00 00 00 00)
   .custom instance void
   [mscorlib]System.Reflection. AssemblyKeyNameAttribute::.ctor(string)
      = (01 00 00 00 00)

...

   .custom instance void
   [mscorlib]System.Reflection. AssemblyTitleAttribute::.ctor(string)
      = (01 00 0A 57 72 6F 78 20 72 6F 63 6B 73 00 00)       // Wrox rocks..

   .hash algorithm 0x00008004
   .ver 1:0:522:37167
}
```

Просматривая файл, вы сможете обнаружить целый ряд объявлений, которые несколько напоминают объявления типа:

```
   [mscorlib]System.Reflection. AssemblyTitleAttribute::.ctor(string)
      = (01 00 0A 57 72 6F 78 20 72 6F 63 6B 73 00 00)       // Wrox rocks..
```

Значение атрибута `AssemblyTitle`, которое мы перед этим ввели, оказалось сохраненным в маннфесте модуля. Если вы воспользуетесь таблицей перекодировки из шестнадцатеричных значений в кодировку ASCII, то обнаружите, что набор символов, расположенный после '01 00 0A', представляет собой не что иное, как текст 'Wrox rocks', представленный в кодировке ASCII. Начальные байты '01 00' представляют собой двухбайтовый идентификатор, а '0A' определяет длину строки (десять символов). Такая процедура сохранения атрибутов известна под названием 'pickling' (консервирование) — этот термин может встретиться в различных материалах по .NET, имеющихся в web.

В приведенном выше участке кода из файла `AssemblyInfo.cs` было использовано слово `AssemblyTitle`, хотя в коде на IL класс представляется в виде `AssemblyTitleAttribute`. Компилятор C# сначала осуществляет поиск класса с названием `AssemblyTitle`, а затем, если поиск не дал результатов, добавляет слово `Attribute` и осуществляет поиск повторно. Поэтому, независимо от того, набираете ли вы название класса полностью или опускаете последнее слово `Attribute`, в обоих случаях будет сгенерирован один и тот же код. На протяжении настоящей главы это слово будет опускаться.

Сохраненное (законсервированное) в манифесте объявление атрибута подозрительно напоминает объект вместе со его конструктором, а байты, заключенные в скобки,— параметры, передаваемые конструктору.

Теперь, получив общее представление об атрибутах, мы можем дать атрибуту следующее определение:

Атрибут — это класс, в котором могут храниться дополнительные данные внутри модуля, имеющие отношение к данному модулю или к типам, содержащимся в данном модуле.

С учетом того, что атрибут — это класс, а в манифесте атрибут хранится в представленном выше формате, можно пересмотреть определение атрибута, данное выше в настоящей главе:

```
[assembly: AssemblyTitle("Wrox rocks")]
```

Этот синтаксис несколько отличается от обычного синтаксиса C#, поскольку он предполагает использование квадратных скобок, в которые заключается атрибут. Тег `assembly:` определяет область действия данного атрибута (которая будет рассматриваться ниже в данной главе), а в остальной части записи объявляется собственно атрибут. У атрибута `AssemblyTitle` есть конструктор, которому передается только один параметр — строковое значение. Компилятор включает это значение в модуль. Доступ к значению может быть осуществлен с помощью стандартного окна свойств Windows, при просмотре этого модуля с помощью `ildasm` или программным путем с помощью **отражения**.

Кроме простых атрибутов, задающих информацию модуля, в .NET Framework описано около двухсот атрибутов, которые используются в самых разнообразных ситуациях — при отладке, при разработке способов временного контроля, преобразовании в последовательную форму и для многого другого. Мы познакомимся с некоторыми стандартными атрибутами после раздела, посвященного отражению, а затем перейдем к рассмотрению вопроса о том, каким образом программист может расширить возможности .NET с помощью собственных атрибутов.

Отражение

Если вы не знакомы с основами языка программирования Java, то отражения окажутся для вас новой темой, поэтому мы посвятим несколько страниц описанию того, что это такое и каким образом оно может быть использовано.

Отражение позволяет осуществлять просмотр модуля программным путем и получать о нем всю информацию, включая информацию о всех типах объектов, которые в нем содержатся. В этой информации содержатся также и все атрибуты, которые вы добавили к этим типам. Объекты отражения хранятся внутри пространства имен `System.Reflection`.

Помимо получения информации о типах, описанных внутри модуля, у программиста имеется возможность генерировать свои собственные модули и типы с помощью пространства имен `System.Reflection.Emit`. Эта тема кажется не совсем подходящей для введения в C#, но если вас она заинтересует, то некоторую информацию по вопросу создания динамических модулей вы сможете найти в MSDN.

Практикум: поиск атрибутов

Первый пример данного раздела будет посвящен просмотру модуля и выводу перечня всех атрибутов, определенных внутри данного модуля; в результате будет получен список, аналогичный приведенному ранее. Код примеров, использованных в данном разделе, можно загрузить с web-сайта Wrox Press, он располагается в директории `Chapter22/FindAttributes`. Ниже полностью воспроизведен исходный файл:

```
//FindAttributes.cs
// Импортирование типов из System и System.Reflection
using System;
using System.Reflection;

/// <summary>
/// Предназначается для раздела главы 22, которая называется отражение
/// </summary>
public class FindAttributes
{
   /// <summary>
   /// Основная точка входа в выполняемый файл .exe
   /// </summary>
   /// <param name="args">Аргументы командной строки - должны
   /// представлять собой модуль</param>
   static void Main(string[] args)
   {
      // Вывод (при необходимости) информации, касающейся
      // использования программы
      if (args.Length == 0)
         Usage () ;
      else if ((args.Length == 1) && (args[0] == "/?"))
         Usage () ;
      else
      {
         // Загрузка модуля
         string assemblyName = null ;

      // Цикл просмотра аргументов, передаваемых консольному приложению.
      // Просмотр аргументов осуществляется с предположением,
      // что приложению передается полное имя пути, включающее пробелы
      // и представляющее собой несколько разных аргументов - из них
      // просто снова составляется единое имя файла...
```

```
            foreach (string arg in args)
            {
               if (assemblyName == null)
                  assemblyName = arg ;
               else
                  assemblyName = string.Format ("{0} {1}" , assemblyName , arg);
            }

            try
            {
               // Попытка загрузить модуль с данным именем
               Assembly a = Assembly.LoadFrom (assemblyName) ;

               // Теперь осуществляется поиск атрибутов модуля
               // В данном случае параметр игнорируется, поэтому
               // он произвольно задан как true
               object[] attributes = a.GetCustomAttributes(true);

               // Если удалось обнаружить описания каких-либо атрибутов...
               if (attributes.Length > 0)
               {
                  Console.WriteLine ("Assembly attributes for '{0}'..." ,
                                     assemblyName);

                  // то следует вывести их...
                  foreach (object o in attributes)
                     Console.WriteLine (" {0}" , o.ToString ());
               }
               else
                  Console.WriteLine ("Assembly {0} contains no Attributes." ,
                                     assemblyName) ;
            }
            catch (Exception ex)
            {
               Console.WriteLine ("Exception thrown loading assembly {0}...",
                                  assemblyName);
               Console.WriteLine ();
               Console.WriteLine (ex.ToString ());
            }
         }
      }

      /// <summary>
      /// Вывод полезной информации для .exe
      /// </summary>
      static void Usage ()
      {
         Console.WriteLine ("Usage:");
         Console.WriteLine (" FindAttributes <Assembly>");
      }
   }
```

Как это работает

В программе сначала производится проверка наличия параметров, переданных через командную строку, если таковые отсутствуют или пользователь ввел сроку `'FindAttributes /?'`, то происходит вызов метода `Usage`, который выводит простое сообщение об использовании данной команды:

```
if (args.Length == 0)
   Usage () ;
else if ((args.Length == 1) && (args[0] == "/?"))
   Usage ();
```

Далее происходит объединение аргументов командной строки в единую строку. Это необходимо выполнить, поскольку пробелы в названиях директорий весьма распространены, например, 'Program Files', а это приводит к тому, что такой текст из-за пробела воспринимается как два разных параметра. Поэтому все аргументы просматриваются в цикле и преобразуются в одну строку, которая используется в качестве имени модуля, подлежащего загрузке:

```
foreach ( string arg in args )
{
   if ( assemblyName == null )
      assemblyName = arg ;
   else
      assemblyName = string.Format ( "{0} {1}" , assemblyName , arg ) ;
}
```

Затем производится попытка загрузить модуль и извлечь все описанные в этом модуле нестандартные атрибуты:

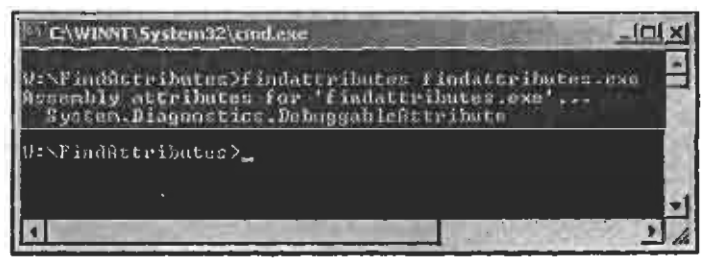
```
Assembly a = Assembly.LoadFrom (assemblyName) ;

// Теперь осуществляется поиск атрибутов модуля В данном случае
// параметр игнорируется, поэтому он произвольно задается как true
object[] attributes = a.GetCustomAttributes(true) ;
```

Все атрибуты, которые удается обнаружить, выводятся на консоль. Теперь можно построить выполняемый файл в Visual Studio.NET или воспользоваться компилятором командной строки:

```
csc FindAttributes.cs
```

В результате этот файл будет откомпилирован, и будет создан выполняемый файл консольного приложения, который вы сможете запустить.



Для того чтобы запустить приложение `FindAttributes`, необходимо задать имя проверяемого модуля. С этой целью здесь можно использовать сам модуль `FindAttributes.exe` (см. рис. справа).

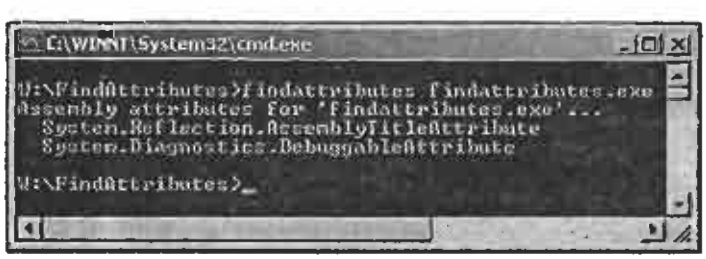
Хотя атрибут `DebuggableAttribute` не был задан, он был включен в модуль компилятором C#. Это обязательно происходит, если выполняемый файл компилируется в отладочном режиме, хотя на момент написания данной книги все выполняемые файлы имеют этот атрибут.

Можно внести в программу изменения, добавив в нее произвольное количество атрибутов модуля. Попробуйте, например, внести в код примера такие изменения:

```
using System;
using System.Reflection;

[assembly: AssemblyTitle("Wrox rocks")]

public class FindAttributes
```



В этом случае, запустив программу, вы сможете увидеть выходной поток, приблизительно тот, что показан на рисунке слева.

Позже мы вернемся к отражению, чтобы продемонстрировать, как можно извлекать атрибуты классов и методов, описанных внутри модуля.

Встроенные атрибуты

.NET Framework включает в модуль несколько атрибутов, например среди них: `DebuggableAttribute` и `AssemblyTitleAttribute`. В настоящем разделе мы обсудим некоторые из наиболее часто употребляемых атрибутов, описанных в .NET Framework, и ситуации, когда у вас может возникнуть желание использовать их.

В данном разделе будут рассматриваться следующие атрибуты:

- `System.Diagnostics.ConditionalAttribute`
- `System.ObsoleteAttribute`
- `System.SerializableAttribute`
- `System.Reflection.AssemblyDelaySignAttribute`

За более подробной информацией об атрибутах, которые поставляются совместно с .NET, следует обратиться к MSDN, либо инсталлировав ее одновременно с инсталляцией .NET, либо воспользовавшись версией, работающей в режиме онлайн по адресу http://msdn.microsoft.com.

Другим чрезвычайно полезным инструментом при работе с .NET, является программа 'Reflector', которую можно переписать с http://www.aisto.com/roeder/dotnet/. В ней для проверки содержимого модуля используются отражения. Она позволяет найти все классы, производные от класса `System.Attribute`, с помощью нескольких щелчков мышью. Это тот инструмент, без которого вы не сможете обойтись.

Атрибут System.Diagnostics.ConditionalAttribute

Это один из наиболее полезных атрибутов, поскольку он позволяет добавлять и исключать участки кода в зависимости от значения флажка, определяемого на этапе компиляции. Рассматриваемый здесь атрибут содержится в пространстве имен `System.Diagnostics`, в которое включены классы, предназначенные для отладки и трассировочного вывода, регистрации событий, счетчики скорости выполнения и информация о процессе. В следующей программе приводится пример использования этого атрибута:

```
using System;
using System.Diagnostics;

class TestConditional
{
   static void Main(string[] args)
   {
      // Создание нового объекта TestConditional
      TestConditional tc = new TestConditional ( ) ;

      // Вызов метода, который доступен, только если определена
      // переменная DEBUG...
      tc.DebugOnly ( ) ;
   }

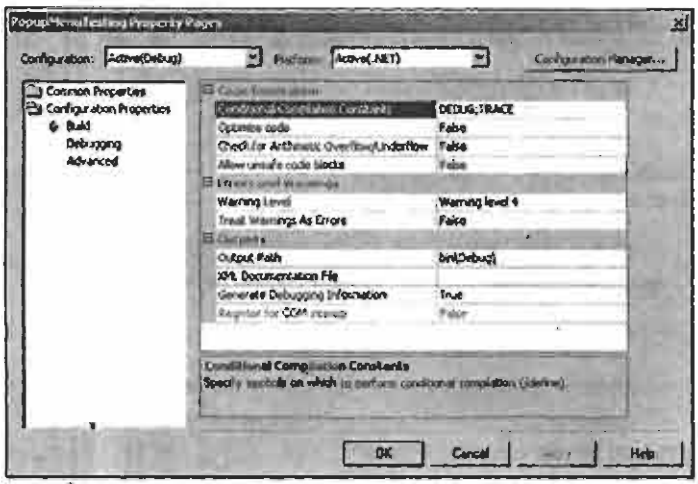
   // Class constructor (Класс "Конструктор"
   public TestConditional ()
   {
   }

   // Этот метод зависит от значения атрибута и будет включен
   // в выходной код только в том случае, если при компиляции
```

```
        // программы был определен флажок DEBUG
        [Conditional("DEBUG")]
        public void DebugOnly ()
        {
            // Эта строка будет выводиться только в случае построения
            // отладочных версий...
            Console.WriteLine ("This string only displays in Debug") ;
        }
}
```

Исходный текст для этого примера найдите в директории `Chapter22/Conditional`. В программе создается экземпляр класса `TestConditional` внутри статической функции `Main`. Затем вызывается метод `DebugOnly`, который зависит от атрибута `Conditional`. Это функция просто выводит текстовую строку на консоль.

При компиляции исходного файла на C# можно определять флажки в командной строке. Атрибут `Conditional` запрещает вызов зависящего от флажка метода при отсутствии соответствующего флажка.



Флажок **DEBUG** будет установлен автоматически, если вы компилируете отладочную версию в Visual Studio.NET. Если вы хотите определить или переопределить какие-либо флажки для конкретного проекта, то необходимо открыть диалоговое окно, позволяющее задавать свойства проекта с помощью опции **Build** в разделе **Configuration Properties**, как показано на рисунке слева.

Заметьте, что по умолчанию при построении отладочной версии предполагается определение флажков **DEBUG** и **TRACE**.

Для определения флажка в командной строке необходимо воспользоваться переключателем `/d:` (сокращение от `/define:` можно при желании набрать всю строку полностью):

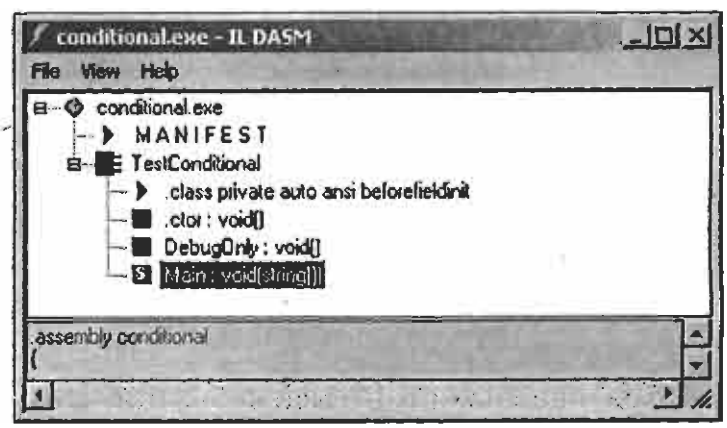
```
csc /d:DEBUG /r:System.dll conditional.cs
```

Если откомпилировать и запустить файл с использованием указанной командной строки, то на консоли можно увидеть следующий выходной поток:

```
This string only displays in Debug
(Это строка выводится только в отладочном режиме)
```

Если откомпилировать программу без определения флага **DEBUG** в командной строке, то программа ничего не выведет.

Для того чтобы стало более понятно, что именно происходит в создаваемом коде, просмотрите его с помощью `ildasm` (см. рис. справа).



Если флажок **DEBUG не задан**, то для метода `Main()` генерируется следующий IL-код:

```
.method private hidebysig static void Main(string[] args) cil managed
{
   .entrypoint
   // Code size 7 (0x7)
   .maxstack 1
   .locals (class TestConditional V_0)
   IL_0000: newobj instance void TestConditional::.ctor()
   IL_0005: stloc.0
   IL_0006: ret
} // end of the method TestConditional::Main
```

Этот код просто создает экземпляр объекта `TestConditional` (`IL_0000`), помещает его в локальную переменную и производит возврат.

Если при компиляции будет задан переключатель `/d:DEBUG`, то будет сгенерирован код, приведенный ниже:

```
.method private hidebysig static void Main(string[] args) cil managed
{
   .entrypoint
   // Code size 13 (0xd)
   .maxstack 1
   .locals (class TestConditional V_0)
   IL_0000: newobj instance void TestConditional::.ctor()
   IL_0005: stloc.0
   IL_0006: ldloc.0
   IL_0007: callvirt instance void TestConditional::DebugOnly()
   IL_000c: ret
} // end of the method TestConditional::Main
```

В этот код дополнительно включены две выделенные строки, в которых происходит вызов метода по условию. Использование `Conditional` приводит к удалению строк, ответственных за вызов метода, но не самого метода.

Атрибут `Conditional` может использоваться только для тех методов, которые ничего не возвращают (описатель `void`), в противном случае удаление обращения к такому методу будет означать, что не возвращено никакого значения. Существует возможность сделать зависимым от атрибута метод, который обладает параметрами типа `ref` или `out`; в этом случае эти переменные просто сохранят свое первоначальное значение.

Атрибут System.ObsoleteAttribute

Включение описания этого атрибута в новую книгу по новому языку программирования может показаться на первый взгляд странным, хотя, на самом деле, это свидетельствует о том, какое внимание разработчики компании Microsoft уделили .NET. С помощью атрибута `Obsolete` может помечаться класс, метод или любое другое вхождение модуля, которое больше не используется.

Этот атрибут может оказаться полезным при создании библиотеки классов. В ходе процесса разработки множества классов некоторые из классов/методов/свойств будут вытесняться новыми. Данный атрибут может быть использован для того, чтобы подготовить пользователей данного кода к фактическому удалению какой-либо конкретной возможности.

Скажем, в версии 10 вашего приложения есть следующий класс:

```
public class Developer
{
   public Developer ()
   {
   }

   public void OriginalMethod ()
   {
   }
}
```

Вы компилировали и использовали этот класс на протяжении нескольких лет, однако теперь появляется нечто более новое:

```
public void NewMethod ()
{
}
```

Вы, естественно, хотите разрешить пользователям вашей библиотеки работать с методом `OriginalMethod()` на протяжении еще некоторого времени, но вы также желаете уведомить их о том, что появился более новый метод, с помощью выдачи предупредительного сообщения в процессе компиляции, информирующего пользователей о существовании метода `NewMethod()`. Все, что для этого потребуется,— это добавить атрибут `Obsolete`, как это показано ниже:

```
[Obsolete("Вместо этого следует использовать метод NewMethod.")]
public void OriginalMethod ()
{
}
```

При компиляции отладочной или окончательной версии компилятор C# выдаст предупрежденне, что вы используете устаревший метод (или метод, который устареет через некоторое время):

```
obsolete.cs(10,3): warning CS0618: 'Developer.OriginalMethod() is obsolete:
(Метод Developer.OriginalMethod() устарел:)
   'Use NewMethod instead'
   ('Вместо этого следует использовать метод NewMethod')
```

С течением времени всем пользователям надоест видеть это предупредительное сообщение при каждой компиляции их программы, и в конечном итоге все пользователи (на самом деле, почти все) перейдут на использование метода `NewMethod()`. Спустя еще какое-то время вы примете решение окончательно отказаться от поддержки метода `OriginalMethod()`, для чего вам потребуется включить в атрибут `Obsolete` дополнительный параметр:

```
[Obsolete("Вместо этого следует использовать метод NewMethod.", true)]
public void CodInC ()
{
}
```

Теперь, если какой-нибудь пользователь попытается откомпилировать этот метод, компилятор сгенерирует ошибку и прекратит процедуру компиляции с выдачей следующего сообщения:

```
obsolete.cs(10,3): error CS0619: 'Developer.OriginalMethod() is obsolete:
(Метод Developer.OriginalMethod() устарел:)
   'You must use NewMethod instead.'
   ('Вместо этого следует использовать метод NewMethod')
```

Использование этого атрибута обеспечивает предоставление помощи пользователям при изменении класса в процессе осуществления модификации их собственных приложений, где этот класс используется.

Для классов, представленных в двоичном виде, например, для компонентов, приобретаемых без исходных текстов, такой способ работы с версиями не может быть рекомендован. В .NET существуют встроенные возможности (обсуждаемые выше) для работы с версиями. Подобные возможности предупреждают, что больше не следует пользоваться какой-либо возможностью созданных нами классов.

Атрибут System.SerializableAttribute

Сериализацией называется процесс сохранения объекта в дисковом файле, памяти или в каком-либо другом месте. В процессе сериализации все данные экземпляра сохраняются на некотором носителе, а в процессе десериализации объект снова наполняется таким образом, что его становится невозможно отличить от первоначального экземпляра.

Все, кому ранее приходилось программировать на MFC, ATL или VB и кто был вынужден постоянно думать о том, каким образом сохранить или считать данные некоего экземпляра, поймут, от какого объема вводимой информации может избавить этот атрибут. Предположим, что у вас имеется некоторый объект, который необходимо сохранить в файле и который описывается следующим образом:

```
public class Person
{
   public Person ()
   {
   }

   public int Age;
   public int WeightInPounds;
}
```

В C# (и в любом другом структурном языке программирования) есть возможность сериализовать члены данного экземпляра без написания какого-либо кода. Все, что вас требуется,— это включить в класс атрибут `Serializable`, а все остальное сделает за вас система выполнения программ .NET.

Когда в названную систему поступает запрос на сериализацию какого-нибудь объекта, то она проверяет, реализован ли в классе данного объекта интерфейс `ISerializable`, и если нет, то имеется ли в этом классе атрибут `Serializable`. Мы не будем обсуждать интерфейс `ISerializable` — это слишком сложная тема для данной книги, к тому же по этой теме уже существует вполне приличная документация в MSDN.

Если в классе обнаруживается атрибут `Serializable`, то .NET использует отражение для того, чтобы получить все данные конкретного экземпляра — независимо от того, описаны ли они как общие, защищенные или частные, и сохранить их как представление этого объекта. Десериализация представляет собой процесс, обратный описанному,— данные считываются с носителя информации и присваиваются переменным экземплярам данного класса.

Ниже приводится пример класса, помеченного атрибутом `Serializable`:

```
[Serializable]
public class Person
{
   public Person
   {
   }
```

```
    public int Age;
    public int WeightInPounds;
}
```

Практикум: сериализация

Полностью код данного примера ищите в поддиректории Chapter22/Serialize. Для того чтобы сохранить экземпляр класса Person, используется объект Formatter, который преобразовывает данные, хранящиеся в классе, в последовательность байтов. Система поставляется с двумя типами таких объектов, используемых по умолчанию,— BinaryFormatter и SoapFormatter (каждый из них обладает своим пространством имен в пространстве имен System.Runtime.Serialization.Formatters). Программа, приведенная ниже, демонстрирует, каким образом можно использовать BinaryFormatter для сохранения объекта класса Person:

```
public static void Serialize ()
{
   // Создание объекта класса Person
   Person me = new Person ();

   // Ввод данных, подлежащих сериализации
   me.Age = 34;
      me.WeightInPounds = 200;

   // Создание на диске файла, предназначенного для хранения данного объекта...
   Stream s = File.Open ( "Me.dat" , FileMode.Create ) ;

   // использование BinaryFormatter для записи этого объекта в поток...
   BinaryFormatter bf = new BinaryFormatter ();

   // Сериализация объекта
   bf.Serialize (s , me);

   // и закрытие потока
   s.Close ();
}
```

Как это работает

Во-первых, в этой программе создается объект класса Person, затем присваиваются значения Age и WeightInPounds, после чего создается поток в виде файла Me.dat, и этот поток закрывается.

Код, выполняющий сериализацию, будет по умолчанию сохранять все содержимое объекта — в большинстве случаев именно это и является необходимым. Однако при определенных обстоятельствах может потребоваться описать одно или несколько полей, не подлежащих сериализации. Этого также несложно добиться:

```
[Serializable]
public class Person
{
   public Person ()
   {
   }

   public int Age;
   [NonSerilized]
   public int WeightInPounds;
}
```

В процессе сериализации такого класса будет сохраняться только член Age, а член WeightInPounds сохраняться не будет и, следовательно, не будет извлекаться при десериализации.

Десериализация, в основном, обратна коду, приведенному выше. Следующий пример открывает поток в виде файла Me.dat, создает BinaryFormatter для считывания объекта и вызывает метод Deserialize, позволяющий извлечь этот объект. После этого считанные данные приводятся к типу Person, и на консоль выводятся возраст и вес:

```
public static void DeSerialize ()
{
   // На этот раз сначала открывается файл
   Stream s = File.Open ("Me.dat" , FileMode.Open);

   // BinaryFormatted используется для чтения объекта (объектов) из потока
   BinaryFormatter bf = new BinaryFormatter ();

   // Десериализация объекта
   object o = bf.Deserialize (s);

   // Убеждаемся в том, что объект нужного нам типа...
   Person p = o as Person ;

   if (p != null)
      Console.WriteLine ("DeSerialized Person aged: {0} weight: {1}" ,
                         p.Age , p.WeightInPounds);

   // Закрытие потока
   s.Close ();
}
```

Обычно посредством NonSerilized (не подлежащие сериализации) помечаются данные, которые получаются в результате вычислений, поскольку они могут вычислены по мере возникновения такой необходимости. В качестве примера можно привести класс, в котором вычисляются простые числа; их вполне можно сохранять для увеличения скорости ответа при обращении к данному классу, однако сериализация и десериализация списка простых чисел не является необходимой, поскольку они всегда могут быть вычислены в момент обращения к классу. В других случаях некоторый член может иметь отношение только к данному экземпляру документа. Например, для объекта, который представляет документ, обрабатываемый редактором, потребуется сериализовать содержимое самого документа, но не текущую позицию, поскольку при очередной загрузке документа в качестве такой позиции просто используется начало документа.

Если необходим более полный контроль за сериализацией документа, то следует реализовать интерфейс ISerializable. Это очень сложная тема, и в настоящей книге ее рассматривать не будем.

Атрибут System.Reflection.AssemblyDelaySignAttribute

В пространстве имен System.Reflection существует несколько атрибутов, уже упоминавшихся в настоящей главе. Одним из наиболее сложных в использовании является атрибут AssemblyDelaySign. В предшествующей главе вы познакомились с тем, каким образом создаются модули и глобальные модули, а также каким образом их можно зарегистрировать в Global Assembly Cache (GAC — кэш глобальных модулей). .NET Framework, кроме того, позволяет помечать модуль как "отсроченный", что, в основном, означает, что вы получаете возможность регистрировать его в GAC для отладочных целей без частного ключа.

Одним из сценариев, при котором возможно использование помеченных таким образом модулей, является разработка коммерческого программного обеспечения. Каждый модуль, разрабатываемый в домашних условиях, должен быть помечен частным ключом вашей компании, прежде чем он будет отправлен клиентам. Поэтому при компиляции модуля вы ссылаетесь на файл с ключом, до того как зарегистрировать модуль в GAC.

Однако многие организации не хотят, чтобы их частный ключ использовался на всех компьютерах разработчиков. По этой причине система выполнения программ позволяет помечать модуль частично, для чего выбирается несколько установок, которые позволяют зарегистрировать модуль в GAC. После тестирования он может быть помечен тем, кто является хозяином файла с ключом. Это может быть информационный отдел вашей компании, одно или несколько доверенных лиц или отдел маркетинга.

В следующем примере продемонстрировано, как можно обычный модуль пометить как "отсроченный", затем зарегистрировать его в GAC для осуществления отладки, после чего окончательно пометить добавлением частного ключа.

Практикум: создание файла с ключом

Создание файла с ключом

Прежде всего необходимо создать файл с ключом с помощью утилиты `sn.exe`, которая обычно инсталлируется в директории `Program Files\Microsoft.NET\FrameworkSDK\Bin`. В этом файле будут храниться общие и частные ключи, поэтому мы назовем его `Company.Key`:

```
sn -k Company.Key
```

Теперь необходимо извлечь ту часть общего ключа, которая будет использоваться разработчиками:

```
sn -p Company.Key Company.Public
```

Эта команда приведет к созданию файла с ключом `Company.Public`, в котором будет содержаться только общедоступная часть ключа. Этот файл с общим ключом может быть скопирован на все компьютеры и не требует каких-то мер по обеспечению безопасности — оберегать следует файл с частным ключом. Спрячьте файл `Company.Key` подальше, поскольку он понадобится, только когда вы захотите окончательно пометить свои модули.

Для того чтобы пометить модуль как "отсроченный" и зарегистрировать его в GAC, необходимо также получить маркер общего ключа, который обычно представляет собой укороченный вариант общего ключа, используемого для регистрации модулей. Получить маркер можно одним из следующих способов:

- Непосредственно из общего файла с ключа
  ```
  sn -t Company.Public
  ```
- Из любого модуля, помеченного этим ключом:
  ```
  sn -T <assembly>
  ```

Оба способа позволяют получить хешированную версию общего ключа, причем они чувствительны к регистру. Это будет объяснено более подробно при описании регистрации модуля.

Практикум: создание "отсроченных" модулей

Приведенный ниже пример демонстрирует, каким образом можно с помощью атрибутов пометить модуль как "отсроченный". Этот пример находится в директории Chapter22/DelaySign:

```
using System;
using System.Reflection;

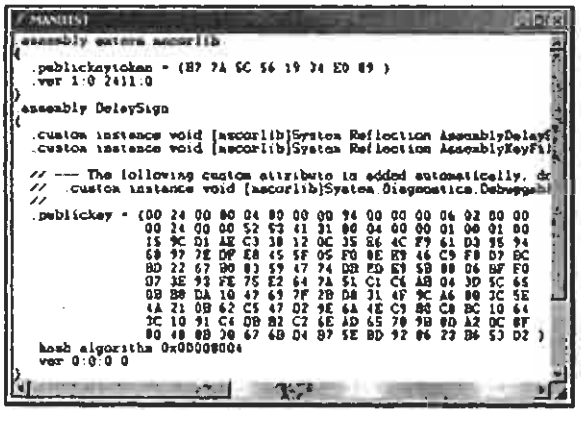
// Определение файла, в котором содержится частный ключ
[assembly: AssemblyKeyFile ("Company.Public")]
// и того факта, что данный модуль должен быть помечен как "отсроченный".
[assembly: AssemblyDelaySign (true)]

public class DelayedSigning
{
   public DelayedSigning ()
   {
   }
}
```

Как это работает

Атрибут `AssemblyKeyFile` позволяет определить, в каком файле следует искать ключ. Это может быть либо файл, содержащий общий ключ, либо — для наиболее доверенных лиц — файл, в котором содержатся как общий, так и частный ключи. Атрибут `AssemblyDelaySign` определяет, будет ли данный модуль помечен обычным образом (`false`) или как "отсроченный" (`true`).

Если для всех проектов вы используете Visual Studio.NET, то в таком случае создается файл с именем `AssemblyInfo.cs`, в котором описываются различные атрибуты, в частности, атрибуты, предназначенные для работы с номерами версий и для хранения информации о файле, которые рассматривались в этой главе. Кроме того, именно в данном файле определяются атрибуты `AssemblyDelaySign`, при этом для каждого нового проекта создаются новые значения, используемые по умолчанию.



После завершения компиляции модуль будет обладать вхождением для общего ключа, который находится в манифесте. На самом деле, в манифесте будет еще достаточно свободного места и для общего ключа, поэтому повторная пометка модуля не приведет к каким-либо существенным изменениям (кроме записи нескольких дополнительных байтов в манифест).

В нижней части экрана (см. рис. слева) представлен общий ключ, хранящийся в манифесте. Необходимо отметить, что в верхней части экрана располагается ссылка на `mscorlib.dll`. В этой библиотеке содержится большая часть объектов пространства имен `System` (и его производных). При этом выводится не только информация о версии данного модуля, но и маркер его общего ключа, который представляет собой последовательность байтов. Если модули сторонних производителей ссылаются на ваш модуль, в них также должна указываться хешированная версия вашего общего ключа.

Если для просмотра модуля, ссылающегося на ваш модуль, используется `ildasm`, то вы увидите в манифесте вхождение, в котором содержится маркер общего ключа.

Для того чтобы добавить ссылку на другой модуль из командной строки, следует воспользоваться опцией /r, как, например, в /r:System.Data.dll:

```
.asembly extern DelaSign
{
   .publickeytoken = (34 AA A4 14 6E E0 1E 4A)
   .ver 0:0:0:0:
}
```

Регистрация в GAC

Попытка использования инструмента gacutil, который находится в директории FrameworkSDK\Bin, для регистрации "отсроченного" модуля приведет к следующему сообщению об ошибке:

```
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Failure adding assembly to the cache: Strong name signature could not be verified.
(Неудача при попытке включить модуль в кэш: Невозможно проверить правильность
сигнатуры строгого имени.)
Was the assembly built delay-signed?
(Не является ли построенный модуль "отсроченным"?)
```

К этому моменту данный модуль является только частично помеченным, а по умолчанию GAC принимает только те модули, у которых имеется полное строгое имя. Однако можно настроить GAC таким образом, что он не будет использовать утилиту sn для проверки правильности строгих имен для "отсроченных" модулей. Помните маркер общего ключа, о котором мы упоминали ранее? Именно здесь он начинает использоваться.

```
sn -Vr*,34AAA4146EE01E4A
```

Эта инструкция разрешает GAC регистрировать все модули, обладающие маркером общего ключа, равным 34AAA4146EE01E4A. Если вы наберете это в командной строке, то получите следующее сообщение:

```
Microsoft (R) .NET Framework Strong Name Utility. Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Verification entry added for assembly '*,34AAA4146EE01E4A'
(Проверка правильности вхождения, включенного в модуль '*,34AAA4146EE01E4A')
```

Теперь попытка инсталлировать модуль в GAC увенчается успехом. Не требуется использовать значение общего ключа при добавлении контрольного вхождения в модуль — можно задать, чтобы все модули были зарегистрированы при использовании следующей строки:

```
sn —Vr *
```

Либо можно однозначно определить модуль, введя его полное имя:

```
sn —Vr DelaySign.dll
```

Такая информация постоянно хранится в так называемой 'Verification Skip Table' (проверочной таблице), которая представляет собой некий файл на диске. Для того чтобы получить список вхождений, хранящихся в этой таблице, необходимо набрать следующее:

```
C:/>sn -Vl
```

Все эти команды чувствительны к регистру. Вот каким будет выходной поток:

```
Microsoft (R) .NET Framework Strong Name Utility. Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Модуль/Строгое имя                Пользователи
=================================================
*,03689116d3a4ae33                Все пользователи
*,33aea4d316916803                Все пользователи

*,34AAA4146EE01E4A                Все пользователи
*,631223CD18E5C371                Все пользователи
*,b03f5f7f11d50a3a                Все пользователи
*,b77a5c561934e089                Все пользователи
```

Обратите внимание на столбец Пользователи — здесь можно определить, чтобы данный модуль загружался в GAC только конкретными пользователями. Дополнительная информация, а также другие возможности при именовании модулей могут быть получены из документации по `sn.exe`.

Завершение строгого имени

Последним этапом процесса является включение общего и частного ключей в модуль с последующей компиляцией. Про модуль с обоими ключами, говорят, что он обладает строгим именем; такой модуль может быть зарегистрирован в GAC без необходимости обходить проверку правильности. Для этого следует воспользоваться утилитой `sn.exe`, на этот раз — с переключателем –R. Наличие переключателя –R означает, что мы хотим заново пометить модуль и добавить новую составную часть в частный ключ:

```
sn -R delaysign.dll Company.Key
```

В результате будет получено следующее:

```
Microsoft (R) .NET Framework Strong Name Utility. Version 1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Assembly 'delaysign.dll' successfully re-signed
(Модуль 'delaysign.dll' помечен заново.)
```

В качестве остальных параметров используются имя модуля, которое необходимо заново пометить, и файл, где содержатся частный и общий ключи.

Нестандартные атрибуты

В первой половине главы мы сосредоточились на некоторых атрибутах, которые входят в состав .NET Framework. Но это еще не все — можно создавать свои собственные атрибуты.

В этой главе мы только обрисуем в общих чертах те возможности, которые таят в себе нестандартные атрибуты. В этом разделе мы познакомимся с разрабатываемыми атрибутами:

- `TestCaseAttribute` позволяет привязать код, в котором осуществляется тестирование некоторого класса, к самому классу
- `BugFixAttribute` позволяет сохранять информацию о том, кто изменял что-либо в исходном коде и когда это делалось
- `DatabaseTableAttribute` демонстрирует, как можно создавать схему базы данных из классов .NET.

Нестандартный атрибут — это просто специальный класс, который должен удовлетворять следующим двум требованиям:

- Атрибут должен быть производным от `System.Attribute`
- Конструктор(ы) атрибута может содержать в себе только типы, которые могут быть найдены в процессе компиляции, например, `string` и `integer`

Эти ограничения, налагаемые на типы, которые могут содержаться в конструкторе (конструкторах) атрибута, возникают вследствие способа, используемого для хранения атрибутов в метаданных модуля. Когда вы применяете атрибут внутри некоторой программы, то вы, фактически, используете конструктор атрибута. Например:

```
[assembly: AssemblyKeyFile ("Company.Public") ]
```

Этот атрибут хранится в метаданных модуля в виде инструкции, вызывающей конструктор атрибута `AssemblyKeyFileAttribute`, которому в качестве параметра передается строка. В приведенном выше примере это строка `"Company.Public"`. Если мы определяем нестандартный атрибут, то пользователи данного атрибута обычно записывают параметры в конструктор класса.

Рассмотрим в качестве первого примера атрибут `TestCaseAttribute`, который показывает, каким образом классы, предназначенные для тестирования могут быть привязаны к тестируемому ими коду.

Атрибут TestCaseAttribute

В программном обеспечении, предназначенном для тестирования, распространенной практикой является определение множества тестовых классов, использующих тестируемый класс (классы), чтобы убедиться в том, что он функционирует надлежащим образом. Это особенно справедливо в случае регрессионного тестирования, когда исправляется некоторая ошибка или добавляется какая-либо новая функциональная возможность и необходимо убедиться в том, что вы ничего не нарушили.

При работе с заказчиками, деятельность которых каким-либо образом регулируется (например, при создании программного обеспечения для фармацевтических компаний, которые работают под жестким контролем государственных агентств), необходимо создавать перекрестные ссылки между основным и трестирующим программным обеспечением. Описываемый в разделе атрибут `TestCaseAttribute` может оказаться полезным для отслеживания связей между неким классом и классом, предназначенным для его тестирования.

Практикум: создание класса нестандартного атрибута

Полностью код данного примера можно найти в директории `Chapter22/TestCase`.

Для создания класса нестандартного атрибута, требуется выполнить следующие действия:

- Создать класс, производный от `System.Attribute`
- Создать необходимые конструктор(ы) и общие свойства
- Включить в этот класс атрибут, который позволит определить, в каких случаях допустимо использовать нестандартный атрибут.

1. Создание класса нестандартного атрибута

Этот шаг оказывается наиболее простым. Все что требуется для его выполнения — это создать класс, производный от класса `System.Attribute`:

```
public class TestCaseAttribute : Attribute
)
{
```

2. Создание конструкторов и свойств

Как отмечалось выше, при использовании атрибута происходит вызов конструктора атрибута. В данном случае нужно определить тип объекта, используемого для проверки класса. Это будет `System.Type`:

```
public class TestCaseAttribute : Attribute
{
   /// <summary>
   /// Конструктор класса
   /// </summary>
   /// <param name="testCase">Объект, который содержит в себе
   /// вариант тестирования</param>
   public TestCaseAttribute (System.Type testCase)
   {
      TestCase = testCase;
   }

   /// <summary>
   /// Выполнение тестирования
   /// </summary>
   punlic readonly System.Type testCase)

   /// <summary>
   /// Проверочный класс
   /// </summary>
   public void Test ()
   {
      // Создание экземпляра тестируемого класса
      // Предполагается, что созданный объект с данным вариантом
      // тестирования будет использоваться для тестирования
      // в конструкторе объекта
      object o = Activator.CreateInstance (TestCase);
   }
}
```

В данной программе описывается единственный конструктор и доступный в режиме только чтение член `TestCase`. Метод `Test()` используется для создания экземпляра с вариантом тестирования, поскольку в данном случае тестирование будет выполняться внутри конструктора тестирующего класса.

3. Включение атрибута в класс для его использования

Последнее, что необходнмо сделать,— это включить атрибут в класс, что позволит указать область примененя этого класса атрибута. Мы хотим объявить, что атрибут варианта тестирования допустимо использовать только в классах. Мы имеем возможность самостоятельно определять, где допустимо использовать создаваемые нами атрибуты. Подробнее этот вопрос будет рассматриваться ниже:

```
[AttrributeUsage(AttributeTargets.Class,
                 AllowMultiple=false,
                 Inherited=true)]
```

```
public class TestCaseAttribute : Attribute
{
   /// <summary>
   /// Конструктор класса
   /// </summary>
   /// <param name="testCase">Объект, который содержит в себе
   /// вариант тестирования</param>
   public TestCaseAttribute ( System.Type testCase )
   {
      TestCase = testCase;
   }


   /// <summary>
   /// Проверочный класс
   /// </summary>
   public readonly SystemType. TestCase:

   /// <summary>
   /// Выполнение тестирования
   /// </summary>
   public void Test ( )
   {
      // Создание экземпляра тестируемого класса
      // Предполагается, что созданный объект с данным вариантом
      // тестирования будет использоваться для тестирования
      // в конструкторе объекта
      object o = Activator.CreateInstance (TestCase);
   }
}
```

Атрибут `AttrributeUsage` обладает единственным конструктором, которому передается значение перечислимого типа `AttributeTargets` (будет полностью описано в этом разделе ниже). В данном случае мы утверждаем, что единственным допустимым местом для размещения атрибута `TestCase` является класс. В этом перечислимом типе можно задавать несколько различных значений с помощью ключевого слова `or` — т. е. допускается использовать другие атрибуты либо в классах, либо в конструкторах, либо в свойствах.

В определении атрибута были использованы и его свойства — `AllowMultiple` и `Inherited`. Эти свойства также будут обсуждаться более подробно в настоящем разделе ниже.

Теперь нам потребуется объект, который будет тестироваться с помощью данного варианта тестирования. В этом классе нет ничего загадочного:

```
[TestCase (typeof(TestAnObject))]
public class SomeCodeOrOther
{
   public SomeCodeOrOther ()
   {
   }

   public int Do ()
   {
      return 999 ;
   }
}
```

Для этого класса в качестве префикса используется атрибут `TestCase`, а далее применяется функция `typeof()` для описания класса, который будет использоваться для тестирования рассматриваемого кода. В данном примере не хватает класса,

осуществляющего тестирование. Ниже приводится объект, используемый для проверки экземпляра кода, который подвергается тестированию:

```
public class TestAnObject
{
   public TestAnObject ()
   {
      // Проверка тестируемого класса
      SomeCodeOrOther scooby = new SomeCodeOrOther () ;

      if (scooby.Do () != 999)
         throw new Exception ("Pesky Kids") ;
   }
}
```

Данный класс просто создает экземпляр тестируемого класса, вызывает метод и генерирует исключительную ситуацию, если возвращенное значение отличается от ожидаемого. Более развернутый вариант тестирования мог бы осуществлять проверку тестируемого объекта более полно, вызывая все методы, которыми обладает данный класс, передавая им значения, выходящие за пределы допустимого для проверки условий возникновения ошибок, и, возможно, создавая какие-либо другие классы в зависимости от текущей информации; например, если тестируемый класс осуществляет доступ к базе данных, то вы, возможно, захотите передать ему в качестве объекта соединение с базой данных.

Перейдем к основному коду. Этот класс в цикле просматривает все типы, определенные в модуле, пытаясь обнаружить те их них, где есть описание атрибута `TestCaseAttribute`. В случае обнаружения такого описания атрибут извлекается, и вызывается метод `Test()`:

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Class,AllowMultiple=false,Inherited=true)]
public class TestCaseAttribute : Attribute
{
   // Код, пропущенный для краткости
}

/// <summary>
/// Класс, в котором используется атрибут TestCase
/// </summary>
[TestCaseAttribute(typeof(TestAnObject))]
public class SomeCodeOrOther
{
   // Код, пропущенный для краткости
}
```

```
// Основной класс программы
public class UnitTesting ( )
{
   public static void Main()
   {
      // Поиск всех классов, в которых содержатся варианты тестирования
      // в данном модуле
      Assembly a = Assembly.GetExecutingAssembly () ;

      // Проход в цикле по всем типам, описанным в модуле, и их тестирование
      // в случае необходимости
      System.Type[] types = a.GetExportedTypes () ;
```

```
            foreach (System.Type t in types)
            {
               // Вывод имени типа...
               Console.WriteLine ("Checking type {0}" , t.ToString ()) ;

               // Включает ли данный тип нестандартный атрибут TestCaseAttribute?
               object[] atts = t.GetCustomAttributes(typeof(TestCaseAttribute),
                                                     false) ;

               if (1 == atts.Length)
               {
                  Console.WriteLine (" Found TestCaseAttribute: Running Tests") ;

                  // OK, данный класс обладает вариантом тестирования. Запускаем его...
                  TestCaseAttribute tca = atts[0] as TestCaseAttribute;

                  try
                  {
                     // Осуществляем тестирование...
                     tca.Test () ;
                     Console.WriteLine (" PASSED!");
                  }
                  catch (Exception ex)
                  {
                     Console.WriteLine (" FAILED!");
                     Console.WriteLine (ex.ToString ());
                  }
               }
            }
         }
      }
```

Новый участок кода выделен серым цветом. В начале работы программа получает выполняющийся модуль посредством статического метода `GetExecutingAssembly()` класса `Assembly`. Далее вызывается метод `GetExportedTypes` для определения всех типов объектов, которые являются общедоступными в данном модуле.

Для каждого экспортируемого в данный модуль типа проверяется наличие в нем атрибута `TestCase`. Если такой атрибут существует, он извлекается (при этом внутри него создается экземпляр атрибута путем передачи параметров, которые внутри кода передаются конструктору объекта) и происходит вызов метода `Test()`, который и производит тестирование кода.

Если эту программу запустить, то будет выведено следующее:

```
Checking type TestCaseAttrubute
(Проверка типа TestCaseAttrubute)
Checking type SomeCodeOrOther
(Проверка типа SomeCodeOrOther)
   Found TestCaseAttrubute: Running Tests
   (Обнаружен TestCaseAttrubute: Запуск тестирования)
   PASSED!
   (Тесты пройдены!)
Checking type TestAnObject
(Проверка типа TestAnObject)
Checking type UnitTesting
(Проверка типа TestCaseAttrubute)
```

Атрибут System.AttributeUsageAttribute

При определении классов нестандартных атрибутов необходимо определять также тип (типы), где данный атрибут может быть использован. В предыдущем примере использование атрибута `TestCase` был определено только для классов. Для того чтобы определить, где использование данного атрибута является допустимым, следует воспользоваться еще одним атрибутом — `AttributeUsage`.

Ниже приведена наиболее простая форма использования этого атрибута:

```
[AttributeUsage(AttributeTargets.Class)]
```

В качестве единственного параметра используется перечислимый тип, указывающий, где допустимо использовать данный атрибут. Если попытаться использовать атрибут `TestCase` для какого-либо метода, компилятор выдаст сообщение об ошибке. Вот один из вариантов недопустимого использования этого атрибута:

```
public class TestAnObject
{
   [TestCase (typeof(System.String))]      // Недопустимое использование
   public TestAnObject ()
   {
      и т. д. ...
   }
}
```

Это приведет к получению следующего сообщения об ошибке:

```
TestCase.cs(54,4):error CS0592: Attribute 'TestCase' is not valid on this declaration type.
   It is valid on 'class' declarations only.
(TestCase.cs(54,4):ошибка CS0592: Использование атрибута 'TestCase' недопустимо
для данного типа объявления. Он может использоваться только с объявлениями 'class'.)
```

В перечислимом типе `AttributeTargets` описаны следующие члены, которые могут использоваться в различных сочетаниях, объединенных с помощью оператора (|), что позволяет определить то множество элементов, в которых допускается использование данного атрибута:

Значение перечислимого типа AttributeTargets	Описание
`All`	Является допустимым в любом месте модуля.
`Assembly`	Является допустимым для модуля. В качестве примера можно привести атрибут `AssemblyKeyFile`, использовавшийся в начале главы.
`Class`	Является допустимым для использования в определении класса. Это значение использовалось для атрибута `TestCase`. Другим примером может служить атрибут `Serializable`.
`Constructor`	Является допустимым только в конструкторах классов.
`Delegate`	Является допустимым только в делегатах.
`Enum`	Может добавляться в описание перечислимого типа. Одним из примеров такого атрибута является атрибут `System.FlagsAttribute`, который, будучи примененным к некоторому перечислимому типу, определяет, что пользователь имеет возможность использовать побитовый оператор or для объединения различных значений этого перечислимого типа. В перечислимом типе `AttributeTargets` этот атрибут также используется.
`Event`	Является допустимым при определении событий.

Продолжение таблицы

Значение перечисляемого типа AttributeTargets	Описание
`Field`	Может быть размещен в некотором поле, например, во внутренней переменной. В качестве примера можно привести атрибут `NonSerialized`, который использовался нами ранее для объявления того, что некоторое конкретное значение не должно сохраняться при сериализации класса.
`Interface`	Является допустимым для интерфейса. Одним из примеров такого атрибута может служить атрибут `GuidAttribute`, описанный в пространстве имен `System.Runtime.InteropServices`, который позволяет явно определять GUID для данного интерфейса.
`Method`	Определяет, что атрибут может использоваться для метода. Это значение используется, например, атрибутом `OneWay`, описанном в пространстве имен `System.Runtime.Remoting.Messaging`.
`Module`	Определяет, что атрибут допустимо использовать в отсеке. Модуль может состоять из нескольких отсеков с кодом; таким образом, этот атрибут позволяет размещать атрибут в отдельном отсеке, а не во всем модуле.
`Parameter`	Может применяться для некоторого параметра в описании метода.
`Property`	Может быть применен к свойству.
`ReturnValue`	Может быть привязан к возвращаемому значению функции.
`Struct`	Является допустимым в определении структур.

Область действия атрибута

В первом примере, приведенном в настоящей главе, нам встретились атрибуты `Assembly*`, синтаксис которых очень напоминает приведенный ниже:

```
[assembly: AssemblyTitle("Wrox rocks")]
```

Строка `assembly:` определяет область действия атрибута, которая в данном случае служит для передачи компилятору информации о том, что атрибут `AssemblyTitle` должен применяться к самому модулю. Модификатор области действия должен применяться только в тех случаях, когда компилятор оказывается не в состоянии самостоятельно определить эту область действия. Допустим, что вы хотите включить атрибут в значение, возвращаемое следующей функцией:

```
[MyAttribute ()]
public long DoSomething ()
{
   ...
}
```

Когда компилятор дойдет до этого атрибута, он примет вполне обоснованное предположение о том, что атрибут применяется ко всему методу, а это совсем не то, что требуется в данном случае, поэтому следует воспользоваться модификатором области действия, для того чтобы явно указать, к чему данный атрибут относится:

```
[return:MyAttribute ()]
public long DoSomething ()
{
   ...
}
```

Для определения области действия атрибута следует использовать одно из приведенных ниже значений:

- `assembly` — атрибут применим ко всему модулю
- `field` — атрибут применим к полю перечислимого типа или класса
- `event` — атрибут применим к событию
- `method` — атрибут применим к методу, описанию которого он предшествует
- `module` — атрибут хранится в отсеке
- `param` — атрибут применим к параметру
- `property` — атрибут хранится для данного свойства
- `return` — атрибут применим к возвращаемому функцией значению
- `type` — атрибут применим к классу, интерфейсу или структуре.

Большинство из этих значений используются очень редко, поскольку область действия обычно определяется однозначно. Однако значения `assembly`, `module` и `return` часто приходится использовать для определения области действия. Если есть какая-либо неоднозначность, к чему относится данный атрибут, то компилятор самостоятельно выбирает тот объект, к которому этот атрибут будет приписан. Наиболее распространенным случаем является использование атрибута для возвращаемого значения функции, как показано ниже:

```
[SomeAttribute]
public string DoSomething ();
```

В данном случае компилятор воспримет этот атрибут как относящийся ко всему методу, а не к возвращаемому значению. Для того чтобы добиться желаемого эффекта, необходимо описать область действия атрибута следующим образом:

```
[return:SomeAttribute]
public string DoSomething ();
```

Атрибут AttributeUsage.AllowMultiple

С помощью этого атрибута мы определяем, может ли пользователь включать один или несколько одинаковых атрибутов в некоторый элемент. Например, можно создать атрибут, в котором будет храниться перечень всех исправленных ошибок, применимый к некоторому участку кода. При создании модуля может появиться необходимость включить в него подробное описание нескольких обнаруженных ошибок.

Атрибут BugFixAttribute

В приведенном ниже коде определяется простой атрибут `BugFixAttribute` и используется флажок `AllowMultiple` с тем, чтобы данный атрибут мог быть использован более одного раза для некоторого участка кода:

```
[AttributeUsage (AttributeTargets.Class | AttributeTargets.Property |
                 AttributeTargets.Method | AttributeTargets.Constructor ,
                 AllowMultiple=true)]
public class BugFixAttribute : Attribute
{
   public BugFixAttribute (string bugNumber , string comments)
   {
```

```
        BugNumber = bugNumber;
        Comments = comments;
    }

    public readonly string BugNumber;
    public readonly string Comments;
}
```

Конструктор атрибута `BugFix` принимает иомер ошибки и строку комментария, при этом он помечеи флажком `AllowMultiple=true`, откуда следует, что он может быть использован, например, таким образом:

```
[BugFix("101","Created some methods")]
public class MyBuggyCode
{
    [BugFix("90125","Removed call to base()")]
    public MyBuggyCode ()
    {
    }

    [BugFix("2112","Returned a non null string")]
    [BugFix("38382","Returned OK")]
    public string DoSomething ()
    {
        return "OK" ;
    }
}
```

Синтаксис, использованный для присваивания значения флажку `AllowMultiple`, имеет несколько страиный вид. Конструктор атрнбута `AttributeUsage` принимает только один параметр — перечень флажков, определяющих область применения данного атрибута. `AllowMultiple` является свойством атрибута `AttributeUsage`, следовательно приведенный ниже синтаксис означает "создай атрибут, а затем присвой его свойству `AllowMultiple` значение `true`":

```
[AttributeUsage (AttributeTargets.Class | AttributeTargets.Property |
                 AttributeTargets.Method | AttributeTargets.Constructor ,
                 AllowMultiple=true)]
public class BugFixAttribute : Attribute
{
    ...
}
```

Схожий метод используется и для свойства `Inherited`, которое обсуждается ниже в иастоящей главе. Если нестандартный атрибут обладает какими-либо свойствами, то нм можно присваивать значения аналогичным способом. Примером может служить добавленне фамилии человека, исправившего данную ошнбку:

```
public readonly string BugNumber;
public readonly string Comments;
public string Author = null;

public override string ToString ()
{
    if (null == Author)
        return string.Format ("BugFix {0} : {1}" ,
                              BugNumber , Comments) ;
    else
        return string.Format ("BugFix {0} by {1} : {2}" ,
                              BugNumber , Author , Comments) ;
}
```

В результате этого мы добавляем свойство `Author` (автор) и переопределенную реализацию метода `ToString()`, которая позволяет выводить подробную информацию об ошибке, если свойству `Author` присвоено какое-либо значение, и только номер ошибки и комментарии к ней в противном случае. Метод `ToString()` может быть использован для вывода перечня исправленных ошибок для данного участка кода — например, распечатан и сохранен в файле.

После того как создан атрибут `BugFix`, вам потребуется некий способ, позволяющий сообщать об исправленных ошибках в данном классе и членах данного класса.

Этот способ заключается в передаче типа класса (снова `System.Type`) функции `DisplayFixes`, приведенной ниже. В ней также используется отражение для поиска всех исправленных ошибок для данного класса, а затем эта функция проходит по всем методам этого класса в поисках атрибутов исправленных ошибок.

Этот пример может быть найден в директории `Chapter22/BugFix`:

```
public static void DisplayFixes (System.Type t)
{
   // Получение всех исправленных ошибок для данного типа;
   // в данном случае предполагается что это класс
   object[] fixes = t.GetCustomAttributes (typeof (BugFixAttribute) , false);

   Console.WriteLine ("Displaying fixes for {0}" , t);

   // Вывод информации об исправленной ошибке
   foreach (BugFixAttribute bugFix in fixes)
      Console.WriteLine (" {0}" , bugFix);

   // Теперь требуется получить все члены (т. е. функции) данного класса
   foreach (MemberInfo member in t.GetMembers (BindingFlags.Instance |
                                               BindingFlags.Public |
                                               BindingFlags.NonPublic |
                                               BindingFlags.Static ))
   {
      // И отыскать все исправленные ошибки и в них также
      object[] memberFixes = member.GetCustomAttributes (typeof (BugFixAttribute) ,
                                                         false) ;
      if (memberFixes.Length > 0)
      {
         Console.WriteLine (" {0}" , member.Name);

         // Проход в цикле и вывод этих ошибок
         foreach (BugFixAttribute memberFix in memberFixes)
            Console.WriteLine (" {0}" , memberFix) ;
      }
   }
}
```

Как это работает

Первое, что осуществляется в данном коде, — это извлечение всех атрибутов `BugFix`, относящихся к самому типу:

```
object[] fixes = t.GetCustomAttributes (typeof (BugFixAttribute) , false);
```

Эти атрибуты нумеруются и выводятся. Далее программа проходит по всем описанным членам данного класса посредством использования метода `GetMembers()`:

```
foreach (MemberInfo member in t.GetMembers (
                BindingFlags.Instance | BindingFlags.Public |
                BindingFlags.NonPublic | BindingFlags.Static))
```

Метод `GetMembers()` позволяет извлекать свойства, методы и поля, описанные в данном типе. Для того чтобы ограничить перечень возвращаемых членов, применяется перечислимый тип `BindingFlags` (который описан в пространстве имен `System.Reflection`).

Флажки привязки, передаваемые этому методу, позволяют указать, какие именно члены нас интересуют. В данном случае представляют интерес все переменные — как статические, так и переменные экземпляра, независимо от того, являются ли они видимыми или нет, поэтому мы задаем значения `Instance` и `Static` вместе со значениями `Public` и `NonPublic`.

Получив все интересующие нас члены, мы проходим по ним в цикле, находя все атрибуты `BugFix`, привязанные к данному члену, и выводим их на консоль. Для того чтобы вывести перечень исправленных ошибок для данного класса, все, что требуется,— это обратиться к статическому методу `DisplayFixes()`, передав ему тип класса:

```
BugFixAttribute.DisplayFixes (typeof (MyBuggyCode))'
```

Для класса `MyBuggyCode`, приведенного ранее, это позволит получить следующий выходной поток:

```
Displaying fixes for MyBuggyCode      /Вывод исправлений для класса MyBuggyCode/
  BugFix 101 : Created some methods     /Создание некоторых методов/
  DoSomething
    BugFix 2112 : Returned a non null string       /Возврат ненулевой строки/
    BugFix 38382 : Returned OK                /Возврат OK/
  .ctor
    BugFix 90125 : Removed call to base()    /Удаление обращения к базе данных/
```

Если требуется вывести исправления для всех классов данного модуля, то можно воспользоваться отражением для получения всех типов, имеющихся в модуле, а затем передать каждый из них статическому методу `BugFixAttribute.DisplayFixes`.

Свойство AttributeUsage.Inherited

Любой атрибут можно объявить как наследуемый, присвоив этому флажку значение `true` при определении нестандартного атрибута:

```
[AttributeUsage (AttributeTargets.Class,
                 Inherited = true)]
public class BugFixAttribute : {...}
```

Это позволяет обозначить тот факт, что атрибут `BugFix` будет наследоваться всеми подклассами класса, в котором он используется, что, на самом деле, может быть в одних случаях желательно, а в других — нет. Для атрибута `BugFix` такое поведение, скорее всего, будет нежелательным, поскольку исправленная ошибка обычно относится только к самому классу, а не к производным от него классам.

Предположим, что имеется следующий абстрактный класс, к которому применяется данный атрибут:

```
[BugFix("38383","Made the class abstract")]
public abstract class GenericRow : DataRow
{
   public GenericRow (System.Data.DataRowBuilder builder)
      : base (builder)
   {
   }
}
```

Если мы создадим подкласс данного класса, то, скорее всего, не захотим, чтобы тот же самый атрибут `BugFix` был отражен в этом подклассе, поскольку такая ошибка в этом подклассе не исправлялась. Однако если бы мы описывали семейство атрибутов, которые привязывают члены класса к полям таблицы в базе данных, то в этом случае, возможно, потребовалось бы наследование этих атрибутов.

Чрезвычайно распространенной является практика, когда при описании схемы базы данных используется набор стандартных столбцов, который входит в состав большинства таблиц, таких, например, как `Name` (имя) или `Description` (описание). Можно использовать эти поля при кодировании базового класса, а затем включить в него нестандартный атрибут, который привяжет свойство, описанное в классе, к соответствующему столбцу в базе данных. В других подклассах при необходимости могут использоваться дополнительные поля.

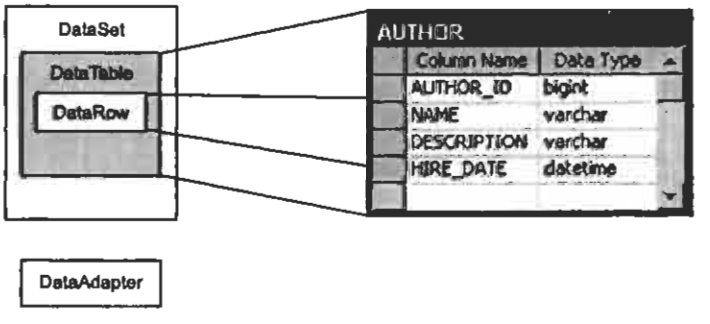
В следующем примере мы создадим атрибуты `DatabaseTable` и `DatabaseColumn`, которые могут быть применены для класса таким образом, чтобы стало возможно автоматическое создание таблицы базы данных, пригодной для хранения данного класса.

Практикум: создание таблиц баз данных с использованием атрибутов

Последний пример данной главы позволит продемонстрировать, каким образом атрибуты могут быть использованы в классах .NET для создания схемы базы данных — структуры базы данных, состоящей из таблиц, столбцов и типов данных, которая позволила бы объектам .NET создавать свои собственные таблицы базы данных, где могли бы храниться эти объекты. Мы познакомимся с тем, как извлекать информацию об этой схеме для создания SQL-запроса, позволяющего создавать таблицы в базе данных, а также создавать объекты `DataTable`, хранящиеся в памяти.

Как стало известно из главы 19, объекты `DataSet`, `DataTable`, `DataRow` и `DataAdapter` используются для осуществления доступа к данным в ADO.NET. Представляется важным использовать эти объекты в соответствии со структурой, на которой основывается база данных. Если в БД со временем вносятся какие-либо изменения, то нужны гарантии того, что эти изменения, например, включение новых столбцов, будут отражены в классах, ответственных за организацию доступа к этой базе данных.

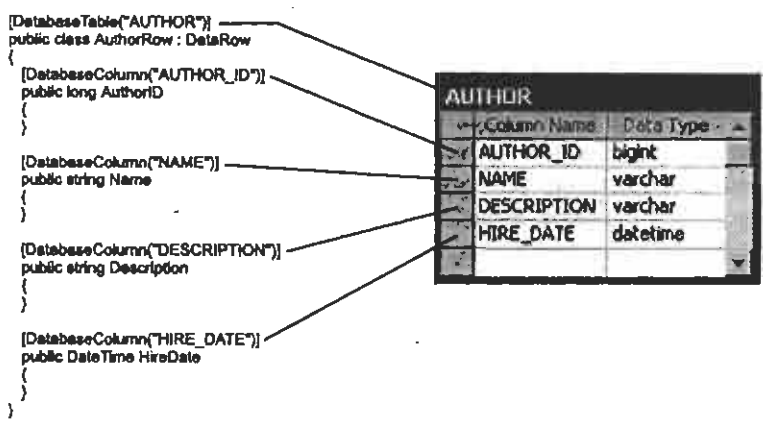
В данном пример мы создадим подклассы класса `DataRow`, предназначенные специально для хранения данных из конкретной таблицы базы данных. Когда основополагающая схема базы данных изменяется не слишком часто, такой подход позволяет обеспечить весьма эффективный способ организации доступа к БД. Если же схема базы данных изменяется достаточно часто или если вы позволяете пользователям самостоятельно вносить изменения в структуру базы данных, то наиболее правильным подходом будет динамическое создание таблиц данных посредством запроса в соответствующей базе данных информации о схеме и построение таблиц данных по ходу работы.



На диаграмме (см. рис. слева) проиллюстрированы отношения между классами ADO.NET и лежащей в их основе таблицей базы данных.

Класс `DataSet` состоит из одного или нескольких объектов `DataTable`, каждый из которых обладает объектами `DataRow`,

которые соответствуют отдельной строке из таблицы БД. Класс `DataAdapter` используется для извлечения данных из базы данных и переноса этих данных в объект `DataTable`, а также для записи данных из объекта `DataTable` обратно в базу данных.



Мы определим базовый объект класса `DataTable`, который будет служить универсальным контейнером объектов `DataRow`. Объекту же `DataRow`, однако, необходимо обеспечить корректный с точки зрения типов доступ к столбцам базы данных, поэтому разобьем его на подклассы. Отношения между этим объектом и лежащей в его основе таблицей, изображены на рисунке слева.

Основное внимание здесь будет уделено таблицам `Books` и `Authors`. Данный пример состоит только из этих двух таблиц, которые будут демонстрироваться на протяжении нескольких следующих страниц. Хотя данный пример разработан для работы с SQL Server, вы можете внести в него изменения, которые позволят вам работать с Oracle или любой другой базой данных.

Класс `AuthorRow` является производным от класса `DataRow` и включает в себя свойства всех столбцов лежащей в его основе таблицы `Author`. К классу `DataRow` был добавлен атрибут `DatabaseTable`, и теперь для каждого свойства, относящегося к конкретному столбцу в данной таблице, существует свой атрибут `DatabaseColumn`. Некоторые параметры этих атрибутов были удалены, чтобы изображение могло уместиться на экране. Со всеми подробностями мы познакомимся в последующих разделах.

Атрибут DatabaseTable

Первый атрибут, который будет использоваться в данном примере, предназначается для передачи информации классу — в данном случае `DataRow` — об имени таблицы базы данных, в которой будет сохраняться объект `DataRow`. Код данного примера может быть найден в директории `Chapter22/DatabaseAttributes`:

```
// Извлечение из файла DatabaseAttributes.cs
/// <summary>
/// Атрибут, который используется в данном классе для определения
/// таблицы базы данных, с которой будет вестись работа
/// </summary>
[AttributeUsage (AttributeTargets.Class , Inherited = false ,
                 AllowMultiple=false)]
public class DatabaseTableAttribute : Attribute
{
   /// <summary>
   /// Создание атрибута
   /// </summary>
   /// <param name="tableName">Название таблицы базы данных</param>
   public DatabaseTableAttribute (string tableName)
   {
      TableName = tableName;
   }
```

```
        /// <summary>
        /// Возвращает название таблицы базы данных
        /// </summary>
        public readonly string TableName;
    }
```

Данный атрибут состоит из конструктора, которому передается имя таблицы в виде строки, и обладает модификаторами `Inherited=false` и `AllowMultiple=false`. Маловероятно, что вы пожелаете, чтобы этот атрибут наследовался какими-либо подклассами, а множественное использование атрибутов запрещено по той простой причине, что данный класс ссылается только одну таблицу.

Внутри класса атрибута название таблицы хранится в виде поля, а не в виде свойства. Это вопрос личного выбора. В данном случае не предусмотрено никакого метода, позволяющего изменять значение поля, в котором хранится название таблицы, поэтому наиболее подходящим является поле, доступное в режиме "только чтение". Если вам больше нравится использовать свойства, то вы можете внести в исходный код соответствующие изменения.

Атрибут DatabaseColumn

Данный атрибут предназначен для использования с общими свойствами класса `DataRow`, он применяется, когда нужно определить названия столбца, к которому данное свойство будет привязано, а также, может ли в данном столбце хранится значение `null`:

```
// Извлечение из файла DatabaseAttributes.cs
/// <summary>
/// Этот атрибут используется для всех свойств, которые
/// представляют столбцы базы данных
/// </summary>
[AttributeUsage (AttributeTargets.Property , Inherited=true ,
                 AllowMultiple=false)]
public class DatabaseColumnAttribute : Attribute
{
    /// <summary>
    /// Создание атрибута столбца базы данных
    /// </summary>
    /// <param name="column">Название столбца</param>
    /// <param name="dataType">Тип данных, хранящихся в данном столбце</param>
    public DatabaseColumnAttribute (string column , ColumnDomain dataType)
    {
        ColumnName = column;
        DataType = dataType;
        Order = GenerateOrderNumber ();
    }

    /// <summary>
    /// Возвращает название столбца
    /// </summary>
    public readonly string ColumnName;
    /// <summary>
    /// Возвращает домен столбца
    /// </summary>
    public readonly ColumnDomain DataType;
    /// <summary>
    /// Считывает/записывает значение флажка Nullable. Возможно,
    /// лучше использовать свойство
    /// </summary>
    public bool Nullable = false;
    /// <summary>
```

```
    /// Считывает/записывает значение порядкового номера Order. В этом
    /// случае использование свойства также может оказаться более приемлемым.
    /// </summary>
    public int Order;
    /// <summary>
    /// Считывает/записывает значение Size, представляющее размер столбца
    /// (полезно для столбцов с текстом).
    /// </summary>
    public int Size;
    /// <summary>
    /// Генерирует возрастающую последовательность порядковых номеров
    /// для столбцов
    /// </summary>
    /// <returns></returns>
    public static int GenerateOrderNumber ()
    {
       return nextOrder++;
    }

    /// <summary>
    /// Частное значение, используемое при генерировании порядковых номеров
    /// </summary>
    private static int nextOrder = 100;
}

/// <summary>
/// Пронумерованный список типов данных в столбцах
/// </summary>
public enum ColumnDomain
{
    /// <summary>
    /// 32 разряда
    /// </summary>
    Integer,
    /// <summary>
    /// 64 разряда
    /// </summary>
    Long,
    /// <summary>
    /// Столбец, содержащий строку
    /// </summary>
    String,
    /// <summary>
    /// Столбец, содержащий дату и время
    /// </summary>
    DateTime
}
```

Данный класс также помечен флажком `AllowMultiple=false`, поскольку в данном случае также имеется соответствие один к одному между свойством `DataRow` и столбцом, к которому оно привязано.

Мы обозначили этот атрибут как наследуемый, с тем чтобы иметь возможность создать нерархию классов для строк базы банных, поскольку существует большая вероятность того, что нам встретятся несколько похожих столбцов в каждой из таблиц, входящих в схему.

Конструктор атрибута приниимает два аргумента. Первый из них представляет собой название столбца, который должен быть определен в базе даиных. В качестве второго аргумента используется значение перечислимого типа `ColumnDomain`, который для даниого примера состоит из четырех значений, чего было бы явно недостаточно для настоящей программы.

У атрибута также есть три другнх свойства, которые описание которых приводится ниже:

- `Nullable` — По умолчанию этому свойству прнсваивается значение `false`; данное свойство используется при создаинн столбца для описания того, может ли в качестве соответствующего значения в базе данных храниться значение `NULL`.
- `Order` — Определяет порядковый номер столбца в данной таблице. После того как таблица будет создана, столбцы будут выводиться в возрастающем порядке. По умолчанию генерируется увеличенное на единицу значение; это происходит виутри конструктора. Можно переопределнть это значение так, как это необходимо.
- `Size` — Определяет максимально допустимое число символов в строке для строкового типа.

Для определения названия столбца `Name` можно воспользоваться таким кодом:

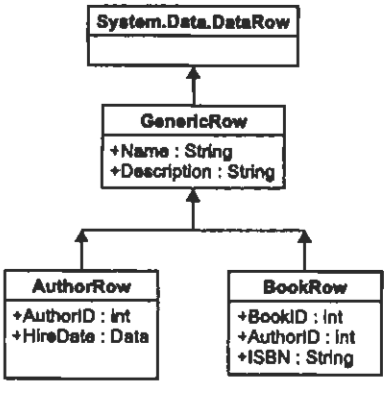
```
[DatabaseColumn("NAME",ColumnDomain.String,Order=10,Size=64)]
public string Name
{
   get { return (string) this ["NAME"]; }
   set { this["NAME"] = value; }
}
```

В этом примере определяется поле с именем `NAME`, и оно будет создано как `VARCHAR(64)`, поскольку домен данного столбца определен как строка, а параметру `Size` присвоено соответствующее значение. Порядковый номер в примере определяется равным 10 — почему именно 10, мы поймем позже. В столбце также не могут использоваться значения `NULL`, поскольку свойству `Nullable` по умолчанию присваивается значение `false` (т. е. даниый столбец будет создан как `NON NULL`).

Класс `DataRow` представляет собой индексатор, которому в качестве параметра передается значение поля (нли порядковое числительное). В результате возвращается некоторый объект, который приводится к строковому типу, перед тем как быть переданным получателю, приведениому выше.

Создание строк баз данных

Целью этого примера является построение семейства строго типизированных объектов `DataRow`. В данном примере мы создадим два класса, `Author` и `Book`, которые являются производнымн от одного базового класса, поскольку они оба обладают доступом к общим полям.



В классе `GenericRow` (см. рис. слева) определяются свойства `Name` н `Description`; необходнмый для этого код прнводится ннже. Данный класс является производиым от класса `DataRow` — класса, являющегося базовым классом для всех строк баз данных, использующнхся в системе.

В нашем примере два класса являются производными от класса `GenericRow`, один из которых представляет автора (`AuthorRow`), а второй — книгу (`BookRow`). Оба класса обладают дополнительными свойствами, которые привязываются к полям базы данных:

```
// Извлечение из файла DatabaseAttributes.cs
/// <summary>
/// Базовый класс строки — в нем описываются
/// столбцы Name и Description
/// </summary>
```

```
public abstract class GenericRow : DataRow
{
   /// <summary>
   /// Создание объекта
   /// </summary>
   /// <param name="builder">Передается из System.Data</param>
   public GenericRow (System.Data.DataRowBuilder builder)
      : base (builder)
   {
   }

   /// <summary>
   /// Столбец, предназначенный для названия записи
   /// </summary>
   [DatabaseColumn("NAME",ColumnDomain.String,Order=10,Size=64)]
   public string Name
   {
      get { return (string) this["NAME"]; }
      set { this["NAME"] = value; }
   }

   /// <summary>
   /// Столбец, предназначенный для описания, который допускает
   /// в качестве значения NULL
   /// </summary>
   [DatabaseColumn("DESCRIPTION",ColumnDomain.String,Nullable=true,
                  Order=11,Size=1000)]
   public string Description
   {
      get { return (string) this["DESCRIPTION"]; }
      set { this["DESCRIPTION"] = value; }
   }
}
```

Тот факт, что класс является производным от `DataRow`, говорит о необходимости создания конструктора, который будет приниматъ единственный параметр — `DataRowBuilder`. Этот класс находится в модуле `System.Data`.

Далее описываются два свойства — `Name` и `Description`, и для каждого из них задаются соответствующие атрибуты:

```
[DatabaseColumn("NAME",ColumnDomain.String,Order=10,Size=64)]
```

В этой строке описывается столбец с именем `NAME`, определяется его домен как строка, имеющая размер 64 символа, и задается его порядковый номер, равный 10. Это сделано таким образом, поскольку при создании таблиц баз данных автор предпочитает, чтобы поля с основными ключами всегда выводились перед всеми остальными полями, содержащимися в данной таблице. Присваивание значения, равного 10, оставляет возможность для включения большого количества идентифицирующих полей. При количестве полей в основном ключе, большем 10, потребует проводить разработку таблицы заново!

Столбцу описания также присваивается название, домен и размер. Свойству `Nullable` присваивается значение `true`, поэтому от нас не требуется обязательного определения столбца описания. Другой возможностью является определение свойства, которому присваивается значение по умолчанию в виде пустой строки, что позволит избежать использование значений `NULL` в базе данных. В качестве порядкового номера столбцу присваивается номер 11, с тем чтобы столбцы с названием и описанием в генерируемой схеме всегда находились рядом:

```
[DatabaseColumn("DESCRIPTION",ColumnDomain.String,Nullable=true,
                Order=11,Size=1000)]
```

В средствах доступа для каждого из этих свойств определяются методы get и set, которые являются строго типизированными, в результате чего для столбца со значениями типа String они возвращают также значение типа String:

```
get { return (string) this["DESCRIPTION"]; }
set { this["DESCRIPTION"] = value ; }
```

Здесь наблюдается некоторое дублирование кода, поскольку атрибут описывает название столбца и, следовательно, вы могли бы в этих методах воспользоваться отражением для извлечения содержимого соответствующего столбца. Отражение, однако, не самый эффективный из всех существующих API, а поскольку эти классы используются для организации доступа к основополагающим столбцам, то от них требуется максимальное быстродействие. Для того чтобы добиться максимального быстродействия из этих средств доступа, можно было бы воспользоваться численными индексаторами столбцов, поскольку использование строк предполагает осуществление поиска по хеш-таблице для определения численного значения индекса. Следует быть осторожным при использовании численных индексаторов, поскольку с ними оказывается несколько более сложно работать, особенно если описывается некоторый подкласс.

Строка AuthorRow создается следующим образом:

```
// Извлечение из файла DatabaseTables.cs
/// <summary>
/// Таблица Author, производная от GenericRow
/// </summary>
[DatabaseTable("AUTHOR")]
public class AuthorRow : GenericRow
{
   public AuthorRow (DataRowBuilder builder)
      : base (builder)
   {
   }

   /// <summary>
   /// Поле с основным ключом
   /// </summary>
   [DatabaseColumn("AUTHOR_ID",ColumnDomain.Long,Order=1)]
   public long AuthorID
   {
      get { return (long) this["AUTHOR_ID"]; }
      set { this["AUTHOR_ID"] = value; }
   }

   /// <summary>
   /// Дата заключения контракта с автором
   /// </summary>
   [DatabaseColumn("HIRE_DATE",ColumnDomain.DateTime,Nullable=true)]
   public DateTime HireDate
   {
      get { return (DateTime) this["HIRE_DATE"]; }
      set { this["HIRE_DATE"] = value; }
   }
}
```

В данном случае мы создали подкласс класса GenericRow и добавили в него свойства AuthorID и HireDate. Обратите внимание на порядковый номер, выбранный для столбца Author_ID. Он выбирается таким образом, чтобы этот столбец выводился первым для данной таблицы. У свойства HireDate такой порядковый номер отсутствует, поэтому соответствующее значение генерируется с помощью

атрибута, а все сгенерированные таким образом значения превосходят 100, по этой причине столбцы таблицы будут представлены в следующей последовательности: AUTHOR_ID, NAME, DESCRIPTION и, наконец, HIRE_DATE.

Класс BookRow также является производным от класса GenericRow, поэтому в него включены свойства для названия и описания. В нем определяются новые свойства BookID, PublishDate и ISBN:

```
// Извлечение из файла DatabaseAttributes.cs
/// <summary>
/// Таблица, в которой будут храниться книги
/// </summary>
[DatabaseTable("BOOK")]
public class BookRow : GenericRow
{
   public BookRow (DataRowBuilder builder)
      : base (builder)
   {
   }

   /// <summary>
   /// Столбец с основным ключом
   /// </summary>
   [DatabaseColumn("BOOK_ID",ColumnDomain.Long,Order=1)]
   public long BookID
   {
      get { return (long) this["BOOK_ID"]; }
      set { this["BOOK_ID"] = value; }
   }

   /// <summary>
   /// Автор, написавший данную книгу
   /// </summary>
   [DatabaseColumn("AUTHOR_ID",ColumnDomain.Long,Order=2)]
   public long AuthorID
   {
      get { return (long) this["AUTHOR_ID"] ; }
      set { this["AUTHOR_ID"] = value; }
   }

   /// <summary>
   /// Дата публикации книги
   /// </summary>
   [DatabaseColumn("PUBLISH_DATE",ColumnDomain.DateTime,Nullable=true)]
   public DateTime PublishDate
   {
      get { return (DateTime ) this["PUBLISH_DATE"]; }
      set { this["PUBLISH_DATE"] = value; }
   }

   /// <summary>
   /// ISBN для данной книги
   /// </summary>
   [DatabaseColumn("ISBN",ColumnDomain.String,Nullable=true,Size=10)]
   public string ISBN
   {
      get { return (string) this["ISBN"]; }
      set { this["ISBN"] = value; }
   }
}
```

Создание SQL

После определения свою базы данных настало время написания кода, который сгенерирует схему базы данных на основании этих классов. В данном примере выходной поток выводится на консоль, поэтому мы могли бы перенаправить его в текстовый файл, запустив .exe из командной строки.

Приведенный ниже класс вызывает метод `OutputTable` для всех типов, для которых мы хотим создать таблицу базы данных:

```
public class DatabaseTest
{
   public static void Main()
   {
      OutputTable (typeof (AuthorRow));
      OutputTable (typeof (BookRow));
   }

   public static void OutputTable (System.Type t)
   {
      // Code in full below
   }
}
```

Мы бы могли использовать отражение, чтобы пройти в цикле по каждому классу в этом множестве, проверяя, не выведен ли этот класс из `GenericRow`, и автоматически выводя классы. Для простоты мы внесли в код имя таблиц, которые должны быть сгенерированы: `AuthorRow` и `BookRow`.

Реализация метода `OutputTable()` представлена ниже:

```
// Извлечение из файла DatabaseAttributes.cs
/// <summary>
/// Создание SQL по типу SQL Server для переданного параметра
/// </summary>
/// <param name="t"></param>
public static void OutputTable (System.Type t)
{
   // Извлечение атрибута DatabaseTable из описания типа
   object[] tableAttributes = t.GetCustomAttributes
                                    (typeof (DatabaseTableAttribute) , true) ;

   // Проверка его существования...
   if (tableAttributes.Length == 1)
   {
      // Вывод SQL в случае удачного завершения проверки
      Console.WriteLine ("CREATE TABLE {0}" ,
                  ((DatabaseTableAttribute)tableAttributes[0]).TableName);
      Console.WriteLine ("(");
      SortedList columns = new SortedList ();

      // Проход по всем свойствам
      foreach (PropertyInfo prop in t.GetProperties ())
      {
         // Получение всех описанных атрибутов DatabaseColumnAttribute
         object[] columnAttributes = prop.GetCustomAttributes
                              (typeof (DatabaseColumnAttribute) , true);

         // Если есть DatabaseColumnAttribute,
         if (columnAttributes.Length == 1)
         {
            DatabaseColumnAttribute dca = columnAttributes[0]
                                                as DatabaseColumnAttribute ;
```

```
                // то выполняется преобразование ColumnDomain
                // в тип данных SQL Server
                string dataType = ConvertDataType (dca);

                // Затем этот SQL-столбец включается в отсортированный
                // список — это сделано, чтобы столбцы выводились
                // в порядке возрастания их порядковых номеров
                columns.Add (dca.Order, string.Format ( " {0,-31}{1,-20}{2,8},",
                   dca.ColumnName ,
                   dataType ,
                   dca.Nullable ? "NULL" : "NOT NULL"));
             }
          }

          // Теперь в цикле просматриваются столбцы SortedList
          foreach (DictionaryEntry e in columns)
             // И выводится следующая строка
             Console.WriteLine (e.Value);

          // После чего SQL завершается.
          Console.WriteLine (")");
          Console.WriteLine ("GO");
          Console.WriteLine ();
       }
    }
```

Этот код отражает переданный тип и осуществляет поиск атрибута `DatabaseTable`. Если удается обнаружить атрибут `DatabaseTable`, то на консоль выводится посылка `CREATE TABLE`, в которую включено название таблицы, содержащееся в атрибуте.

Затем мы проходим в цикле все свойства данного типа, для того чтобы найти все атрибуты `DatabaseColumn`. Любое свойство, обладающее таким атрибутом, превращается в столбец генерируемой таблицы:

```
          foreach ( PropertyInfo prop in t.GetProperties ())
          {
             object[] columnAttributes = prop.GetCustomAttributes
                                   (typeof (DatabaseColumnAttribute) , true);
```

Строковое представление столбца создается посредством представленного ниже вызова метода `ConvertDataType()`. Это представление хранится в отсортированном семействе, поэтому столбцы генерируются на основании значения свойства `Order` данного атрибута.

После просмотра в цикле всех атрибутов и создания соответствующих вхождений в отсортированном списке этот список просматривается в цикле, и каждое значение выводится на консоль:

```
          foreach (DictionaryEntry e in columns)
             Console.WriteLine (e.Value);
```

В завершение мы добавляем закрывающую скобку в команду `GO`, что с точки зрения SQL Server означает необходимость выполнить группу операторов и тем самым создать таблицу.

Последняя функция данного модуля преобразовывает значения из перечислимого типа `ColumnDomain` в конкретный тип данных для используемой базы данных. Кроме того, для строковых столбцов мы создаем представление столбца, в которое входит его размер; так, например, свойство `NAME` из начального базового класса создается в виде `VARCHAR(64)`. Такой тип представляет собой массив символов переменной длины, не превышающей 64 символов.

```
// Извлечение из файла DatabaseAttributes.cs
/// <summary>
/// Преобразование ColumnDomain в тип данных SQL Server
/// </summary>
/// <param name="dca">Атрибут столбца</param>
/// <returns>Строковое представление типа данных</returns>
private static string ConvertDataType (DatabaseColumnAttribute dca)
{
   string dataType = null;

   switch (dca.DataType)
   {
      case ColumnDomain.DateTime:
      {
         dataType = "DATETIME";
         break;
      }
      case ColumnDomain.Integer:
      {
         dataType = "INT";
         break;
      }
      case ColumnDomain.Long:
      {
         dataType = "BIGINT";
         break ;
      }
      case ColumnDomain.String:
      {
         // Включение размера строки...
         dataType = string.Format ( "VARCHAR({0})" , dca.Size ) ;
         break ;
      }
   }

   return dataType;
}
```

Для каждого члена перечислимого типа мы создаем строку столбца, предназначенную для SQL Server. В данном примере это приводит к созданию SQL для классов `Author` и `Book`, как показано ниже:

```
CREATE TABLE AUTHOR
(
   AUTHOR_ID            BIGINT              NOT NULL,
   NAME                 VARCHAR(64)         NOT NULL,
   DESCRIPTION          VARCHAR(1000)       NULL,
   HIRE_DATE            DATETIME            NULL,
)
GO

CREATE TABLE AUTHOR
(
   BOOK_ID              BIGINT              NOT NULL,
   AUTHOR_ID            BIGINT              NOT NULL,
   NAME                 VABCHAR(64)         NOT NULL,
   DESCRIPTION          VARCHAR(1000)       NULL,
   PUBLISH_DATE         DATETIME            NULL,
   ISBN                 VARCHAR(10)         NULL.
)
GO
```

Этот SQL может быть запущен для пустой или заранее существующей базы данных SQL Server с целью создания данных таблиц. Созданные классы `DataRow` могут быть использованы для обеспечения безопасного с точки зрения используемых типов доступа к данным, находящиеся в этих таблицах.

Для того чтобы иметь возможность воспользоваться производными от `DataRow` классами, нам необходимо написать определенный код, подобный тому, который приведен ниже. Этот класс переопределяет минимальное множество функций из `DataTable`, а в качестве параметра его конструктору передается тип строки:

```
// Извлечение из файла DatabaseAttributes.cs
/// <summary>
/// Основной класс таблицы с данными
/// </summary>
public class MyDataTable : DataTable
{
   /// <summary>
   /// Создание объекта на базе класса DataRow
   /// </summary>
   /// <param name="rowType">Класс, производный от класса DataRow</param>
   public MyDataTable (System.Type rowType)
   {
      m_rowType = rowType;
      ConstructColumns ();
   }

   /// <summary>
   /// Создание столбцов DataColumn для данной таблицы
   /// </summary>
   private void ConstructColumns ()
   {
      SortedList columns = new SortedList ();

      // Просмотр в цикле всех свойств
      foreach (PropertyInfo prop in m_rowType.GetProperties ())
      {
         object[] columnAttributes = prop.GetCustomAttributes
                                        (typeof (DatabaseColumnAttribute) , true);

         // Если у данного свойства есть атрибут DatabaseColumnAttribute
         if (columnAttributes.Length == 1)
         {
            DatabaseColumnAttribute dca = columnAttributes[0]
                                                  as DatabaseColumnAttribute;

            // Создание DataColumn
            DataColumn dc = new DataColumn (dca.ColumnName , prop.PropertyType);
            // Присваивание значения его флажку, отвечающему за значение null
            dc.AllowDBNull = dca.Nullable ;
            // И включение его во временное семейство столбцов
            columns.Add (dca.Order , dc);
         }
      }

      // Включение столбцов в возрастающем порядке
      foreach (DictionaryEntry e in columns)
         this.Columns.Add (e.Value as DataColumn) ;
   }

   /// <summary>
   /// Вызывается из System.Data
   /// </summary>
   /// <returns>Типы строк, которые содержатся в данной таблице</returns>
```

```
        protected override System.Type GetRowType ()
        {
            return m_rowType;
        }

        /// <summary>
        /// Создание нового DataRow
        /// </summary>
        /// <param name="builder">Передается из System.Data</param>
        /// <returns>Безопасную с точки зрения использования типов
        /// строку DataRow</returns>
        protected override DataRow NewRowFromBuilder (DataRowBuilder builder)
        {
            // Создание нового экземпляра класса типа нашей строки
            return ( DataRow ) Activator.CreateInstance ( GetRowType() ,
                                                    new object[1] { builder });
        }

        /// <summary>
        /// Сохранение типа строки
        /// </summary>
        private System.Type m_rowType;
    }
```

Функция `ConstructColumns()`, вызываемая из конструктора, создает массив `DataColumns` для `DataTable` — в этом случае колонки извлекаются также с помощью отражения. Остальные методы — `GetRowType()` и `NerRowFromBuilder()` — переопределяют соответствующие методы базового класса `DataTable`.

Создав такой производный класс `MyDataTable`, вы получаете простую возможность использовать его в вашей программе. Приведенный ниже пример демонстрирует процедуру добавления двух записей в таблицу `Author`, а затем вывод этих строк в виде XML-файла:

```
DataSet      ds = new DataSet ();
MyDataTable t = new MyDataTable (typeof (AuthorRow));

ds.Tables.Add (t);
AuthorRow   author = (AuthorRow)t.NewRow ();

author.AuthorID = 1;
author.Name = "Me";
author.HireDate = new System.DateTime ( 2000,12,9,3,30,0 ) ;

t.Rows.Add (author);

author = (AuthorRow) t.NewRow ();
author.AuthorID = 2;
author.Name = "Paul";
author.HireDate = new System.DateTime (2001,06,06,23,56,33);

t.Rows.Add (author);

t.DataSet.WriteXml (@".\authors.xml");
```

Если запустить такую программу, то будет получен следующий выходной поток:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Table1>
    <AUTHOR_ID>1</AUTHOR_ID>
    <NAME>Me</NAME>
    <HIRE_DATE>2000-12-09T03:30:00.0000000-00:00</HIRE_DATE>
  </Table1>
```

```
    <Table1>
      <AUTHOR_ID>2</AUTHOR_ID>
      <NAME>Paul</NAME>
      <HIRE_DATE>2001-06-06T23:56:33.0000000+01:00</HIRE_DATE>
    </Table1>
  </NewDataSet>
```

Этот пример позволил продемонстрировать практический способ использования нестандартных атрибутов в программах. Если вы собираетесь встраивать структуру своей базы данных в классы, которые осуществляют доступ к этой базе данных, то это послужит вам хорошей стартовой площадкой. Привязывание таблиц баз данных классам является приемлемым, когда схема вашей базы данных не изменяется слишком часто, но для более динамичной серверной части может оказаться более приемлемым воспользоваться способом, при котором классы, осуществляющие доступ к данным, отодвигаются на один шаг назад от таблиц баз данных, доступ к которым они осуществляют.

При исчерпывающей реализации мы получили бы возможность использовать атрибуты для описания:

- Столбцов с основными ключами
- Ограничений — внешний ключ и проверка
- Версий — номер версии для каждого атрибута столбца и каждого атрибута таблицы позволил бы упростить создание сценариев обновления; фактически, появилась бы возможность полностью проводить обновление посредством использования атрибутов
- Значений столбцов, использующихся по умолчанию

Итоги

В данной главе описывается, что из себя представляют атрибуты, и обсуждаются некоторые из атрибутов, описанных в системе. На самом деле в системе существует намного большее количество атрибутов, и лучшим способом познакомиться с ними является изучение MSDN-документации.

Мы обсудили атрибуты, которые могут храниться внутри модуля с целью обеспечения конечного пользователя детальной информацией (она выводится с помощью диалога свойств файлов), а также воспользовались `ildasm` для знакомства с тем, каким образом атрибуты хранятся внутри модуля. Мы затронули также предмет "отражение", чтобы продемонстрировать, каким образом можно считывать атрибуты программным путем, наконец, мы описали некоторые встроенные атрибуты, такие как `Conditional`, `Obsolete` и `Serializable`.

После этого мы перешли к рассмотрению вопроса о том, как можно создавать нестандартные атрибуты, и обсудили атрибут `AttributeUsage`. Воспользовавшись некоторыми из обсуждавшихся ранее приемов, мы создали атрибут `BugFix`. В завершение мы рассмотрели большой пример, посвященный генерированию информации о схеме базы данных из классов.

23

Основы web-программирования

На протяжении трех последующих глав мы будем изучать программирование для World Wide Web (Всемирная паутина). Это очень большая тема, и в настоящей главе вы получите хорошие знания как о таящихся в ней трудностях, так и об удивительных открывающихся возможностях. В следующей главе мы познакомимся с использованием ASP.NET, однако для начала нам придется потратить некоторое время для рассмотрения того, что было раньше и каким образом возникли Интернет вообще и Всемирная паутина в частности.

В этой главе будут рассматриваться следующие темы:

- Устройство Интернета
- Основы HTML и HTML-форм
- Различия между неизменяемыми и активными web-страницами
- Различия между сценариями клиентской и серверной сторон
- Вопрос о том, почему ASP.NET представляет собой наилучший выбор для разработки современных web-решений

Краткая история Интернета

Интернет впервые был создан в 60-х гг. прошлого века как проект Министерства обороны США. Он получил название ARPANET (Advanced Research Projects Agency Network — сеть Агентства по перспективным исследовательским проектам). Его основной целью было объединение различных компьютеров, разбросанных по Западному побережью США, с тем чтобы они могли взаимодействовать друг с другом даже в случае начала атомной войны. В то время компьютеры были очень редкими и дорогостоящими устройствами, которые могли позволить себе только университеты и правительственные организации. Связав воедино несколько компьютеров, можно было организовать общий доступ к различной информации и данным, что еще больше увеличивало ценность каждого отдельного компьютера. Протокол, предназначенный для работы в данной сети, был разработан таким образом, чтобы быть устойчивым к нарушениям целостности сети. Это было сделано для того, чтобы даже при выходе из строя в результате ядерной атаки одного или нескольких компьютеров, сеть сохраняла бы свою работоспособность.

Результатом разработки ARPANET явилось создание TCP/IP (Transmission Control Protocol/Internet Protocol — протокол управления передачей/межсетевой протокол). Это протокол, позволяющий вести обмен сигналами, в которых содержатся компьютерные данные и информация, по телефонным линиям, по волоконно-

оптическим кабелям и через спутники. Протокол TCP/IP позволяет различным компьютерам обмениваться между собой информацией по сети, и его принятие расчистило дорогу для появления Интернета в том виде, в котором он знаком нам сегодня.

На протяжении 70-х и 80-х гг. Интернет по-прежнему оставался, в основном, областью, в которой работали академические институты и правительственные учреждения. Но по мере того, как компьютеры становились более дешевыми, а персональные компьютеры стали получать большее распространение, Интернет также продолжал развиваться. С появлением в 90-х гг. прошлого века World Wide Web развитие Интернета стало происходить по экспоненциальному закону. На сегодняшний день огромное количество людей, обладающих компьютерами, используют их для доступа к Web, и, как следствие, большая часть современного бизнеса использует Web для продажи или продвижения своих продукции и услуг. Не будет преувеличением сказать, что Интернет в целом и World Wide Web в частности в корне изменили ведение бизнеса.

World Wide Web

Простой вопрос — знаете ли вы, чем отличается Интернет от World Wide Web? Если вы отвечаете "ничем", то вам стоит продолжить чтение. Интернет — это цельная связанная сеть компьютеров, которая охватывает весь земной шар. На миллионах серверов располагаются миллиарды web-страниц, групп электронной почты, дискуссионных страниц и FTP-сайтов. Интернет целиком работает в режиме онлайн, включая электронную почту, FTP, новости, Gopher-сайты, чат-программы и т. п. World Wide Web появилась позже Интернета. Первый web-сервер заработал в 1991 г., но на сегодняшний день Всемирная паутина — это огромная часть Интернета, состоящая из взаимосвязанных web-страниц, в которых содержится текст, графика и мультипликация. Ключевым компонентом World Wide Web являются гиперссылки (текстовые или графические), которые позволяют пользователю переходить на другие web-страницы. Это оказывается возможным благодаря использованию **HyperText Markup Language (HTML, язык гипертекстовой разметки)**, к рассмотренню которого мы вскоре перейдем.

До появления Паутины Интернет в основном базировался на тексте. Паутина представляет собой чрезвычайно дружественную по отношению к пользователям среду для взаимодействия их друг с другом, что оказывается по-настоящему привлекательным для подавляющего большинства людей. Использование Интернета для получения информации перестало быть прерогативой только ученых и специалистов. Помните об этом, когда будете разрабатывать свой очередной сайт! В этой книге мы будем рассматривать исключительно программирование для World Wide Web.

Каким образом работает Паутина

Паутина представляет собой пример клиент-серверной системы. Любой объект в Паутине может быть представлен либо как сервер, либо как клиент.

Серверы

Все, что видите в World Wide Web, находится на каких-либо серверах, разбросанных по всему миру. Сервер — это приложение, выполняющееся на некотором компьютере и обеспечивающее доступ к информации, файлам или данным, которые запрашиваются каким-либо другим лицом, приложением или компьютером.

Если говорить конкретно о web-страницах, то web-сервер отвечает на запросы, касающиеся тех web-страниц, которые хранятся на этом сервере. Клиент посылает запрос на получение web-страницы web-серверу, а web-сервер, в свою очередь, отправляет обратно клиенту страницу для просмотра.

Местоположение серверов определяется с помощью **Uniform Resource Locator (URL — универсальный указатель ресурса)**. Это особый адрес web-страницы, по которому она располагается в World Wide Web. Он указывает на местоположение на каком-либо сервере и на запрашиваемую web-страницу. Например, URL: `http://www.microsoft.com/ms.htm` ссылается на сервер в Интернете с именем `www.microsoft.com` и на web-страницу с именем `ms.htm`, расположенную на сервере. Подобно пути, используемому в файловых системах, URL может содержать в себе названия поддиректорий для более глубокого поиска по содержимому данного сайта.

У каждого web-сайта имеется свой уникальный IP-адрес (Internet Protocol — межсетевой адрес), с помощью которого он может быть идентифицирован. Такой адрес состоит из четырех чисел, которые имеют следующий формат: `200.123.45.1` (числа отделяются друг от друга точками, каждое число может изменяться в диапазоне от 0 до 255). К счастью, нет необходимости держать в памяти IP-адреса web-сайтов, которые вы посещаете; специальные серверы, называемые DNS-серверами (Domain Name Services — служба доменных имен), автоматически выполняют необходимые преобразования URL и IP-адресов. Такие DNS могут существовать как внутри вашей собственной сети, так и вне — в пределах Интернета. Необходимо понимать, что на одном web-сервере может располагаться несколько различных web-сайтов, каждый из которых обладает своим собственным URL. В этом случае IP-адреса всех таких web-сайтов будут ссылаться на один и тот же компьютер, а приложение web-сервера распределит клиентские запросы по соответствующим web-сайтам.

Клиенты

Клиентом называется особая часть программного обеспечения, выполняющаяся на компьютере. Наиболее распространенным клиентом в Интернете является браузер — специальная программа, позволяющая пользователю вводить адреса или использовать гиперссылки на web-страницах для поиска новых web-страниц. Она управляет отправкой запросов web-серверу и выводом на экран полученной от web-сервера web-страницы. На сегодняшний день двумя наиболее распространенными браузерами являются Navigator компании Netscape и Internet Explorer компании Microsoft.

На следующем рисунке приводится пример, каким образом на сервер посылается клиентский запрос и как в ответ web-страница отправляется обратно. Этот ответ представляется в виде HTML-страницы, которая возвращается браузеру клиента:



Протокол HTTP — HyperText Transfer Protocol

С точки зрения пользователя отправка и получение данных осуществляется совершенно прозрачно. Все, что видит пользователь в своем браузере,— это либо выведенная страница, либо сообщение об ошибке, если произошло что-то непредвиденное. Однако разработчику требуется обладать более полным представлением относительно того, как именно происходит такое взаимодействие. Мы уже упоминали TCP/IP — протокол, который отправляет данные по линиям связи и определяет то, каким образом эти данные передаются к месту назначения. Еще одним акроимом, который придется запомнить, является **HTTP**, что расшифровывается как **HyperText Transfer Protocol** (**протокол передачи гипертекста**). Это протокол, который определяет, что именно необходимо сделать с данными. Это прикладной протокол, позволяющий браузерам и серверам взаимодействовать друг с другом и обмениваться между собой данными. Все запросы на получение web-страниц и ответы, отправляемые серверами, представляют собой HTTP-сообщения, которые пересылаются между браузером и сервером.

Типы HTTP-сообщений

Для получения от web-сервера какого-либо ресурса вроде web-страницы клиент должен отправить web-серверу **сообщение с HTTP-запросом**. Такое сообщение включает имя запрашиваемого ресурса (например, имя файла, в котором содержится web-страница), а также некоторые другие данные (называемые заголовками), которые несут в себе дополнительную информацию о самом клиенте. Web-сервер обрабатывает запрос и формирует **сообщение с HTTP-ответом**, предназначенное для данного клиента. Ответ содержит состояние запроса, другие заголовки, которые включаются в ответ, и, наконец, сами запрашиваемые данные. Если по какой-нибудь причине не удается выполнить запрос, то обратно клиенту отправляется сообщение о неудаче, которое включается в информацию о состоянии запроса, но никаких данных в ответе не посылается.

Представляется важным запомнить, что в большинстве случаев вам не будет видно ни одного из этих сообщений. Созданием и отправкой запросов занимается браузер, и он же обрабатывает сообщения с HTTP-ответами, полученные от сервера. Однако очень важно осознавать те разнообразные возможности, которые у вас имеются, и что они означают.

Команда HTTP GET

Наиболее часто встречающимся типом HTTP-запроса является команда `GET` (получить), предназначенная для запроса у сервера некоторого файла с web-страницей или какого-нибудь другого ресурса. Формат команды `GET` очень прост; эта команда состоит из имени ресурса, которые требуется получить от сервера, и из нуля или нескольких строк заголовков. Ниже приводится пример допустимого запроса `GET`:

```
GET /content/booklistings.html HTTP/1.0
From: user@wrox.com
User-Agent: Mozilla/4.0
```

Строка `/content/booklistings.html` определяет тот конкретный ресурс, который запрашивается у web-сервера. Просмотр папок сервера начинается с корневой папки, следовательно, поскольку данный запрос посылается на `www.wrox.com`, он приведет к получению web-страницы, обладающей следующим значением URL:

http://www.wrox.com/content/booklistings.html. Строки, начинающиеся со слов From и User-Agent, представляют собой примеры обычных HTTP-заголовков, которые включаются в состав сообщения-запроса.

Команда HTTP POST

Другая распространенная форма сообщения-запроса, которое может быть отправлено web-серверу,— это команда POST (отправить). POST обычно используется при отправке на сервер информации для ее последующей обработки. Эта команда обычно используется для проверки данных HTML-формы, к чему мы еще вернемся ниже в этой главе. Сообщения HTTP POST выглядят очень похожими на сообщения HTTP GET, хотя наиболее часто запрашиваемый в них ресурс оказывается серверным приложением или активной серверной страницей (активная серверная страница представляет собой особую разновидность web-страницы, которая в динамике генерирует HTML в зависимости от получаемых ею данных). В отличие от сообщения GET наличие в этом запросе полезной информации отправляемой серверу, является обязательным. Ниже приводится пример запроса POST, отправляемого на web-сервер:

```
POST /content/GetBooks.asp HTTP/1.0
From: user@wrox.com
User-Agent: Mozilla/4.0
Content-Type: text-plain
Content-Length: 33

Здесь располагаются данные, предназначенные для сервера.
```

Этот запрос отправляется файлу активной серверной страницы (Active Server Page, ASP), расположенному на сервере, обрабатывающему полученные данные и возвращающему клиенту динамически сгенерированную HTML-страницу. В приведенном выше запросе заданы также два дополнительных HTTP-заголовка — Content-Type и Content-Length. Web-сервер использует эти заголовки при обработке информации, полученной им в команде POST. Данные, которые на самом деле были получены сервером,— это строка "Здесь располагаются данные, предназначенные для сервера.".

Строка запроса

Другим способом передачи данных серверным приложениям является использование команды GET для передачи информации в строке запроса. Это обычный URL-запрос, к которому непосредственно добавлены необходимые данные, что оказывается полезным при передаче серверу небольших объемов информации в текстовом виде. Данные присоединяются к URL в виде пар имя/значение, причем, для того чтобы отделить адрес от данных, используется вопросительный знак, а пары имя/значение отделяются друг от друга символом &. Ниже приводится пример запроса HTTP GET с данными, включенными в него в формате строки запроса:

```
GET /Books/Book_Details.asp?isbn=1861004982&order=title HTTP/1.0
From: user@wrox.com
User-Agent: Mozilla/4.0
```

Эти данные также направляются в ASP-файл вместе с полями isbn и order, включенными в URL. Использование команды GET для отправки данных серверным приложениям оказывается очень простым и легкодоступным, но для него существуют определенные ограничения. Во-первых, так могут передаваться только текстовые данные, и, во-вторых, количество данных в строке запроса может быть очень небольшим. Если необходимо отправить нечто большее, чем просто несколько полей

с данными, то следует воспользоваться командой POST. Кроме того, все данные, расположенные в строке запроса, абсолютно доступны на протяжении всей передачи через установленное соединение, поэтому данные, для которых существует риск с точки зрения обеспечения безопасности (например, информация о кредитных картах или пароли), никогда не следует включать в строку запроса.

HTTP-заголовки

Имя заголовка	Описание
From	Информация, идентифицирующая электронный адрес клиента, которая задается при определении установок браузера
User-Agent	Тип и версия программного обеспечения, которая используется клиентом для отправки HTTP-запроса
Content-Type	Тип данных, содержащихся в теле данного HTTP-сообщения
Content-Length	Длина данных, содержащихся в теле данного HTTP-сообщения
Date	Дата и время формирования данного сообщения

В предшествующих примерах встретилось несколько разновидностей HTTP-заголовков. HTTP-заголовки можно представить себе как обычные заголовки в документах — они не являются страницей или документом, но в них содержатся жизненно важные метаданные о содержимом документа. С каждой HTTP-командой, посылаемой клиенту или серверу, происходит отправка не только данных, но и последовательности заголовков. Каждый из заголовков обычно представляет собой некоторое поименованное свойство; в нем содержится имя заголовка, а за ним его значение. В таблице слева в качестве примера приводятся некоторые из HTTP-заголовков.

Web-программирование 101

Поскольку эта книга посвящена программированию, начнем писать хоть какие-нибудь программы! Мы собираемся рассмотреть процедуру разработки web-страниц, начиная с самой простейшей и постепенно переходя к ASP.NET. Это поможет понять, каким образом web-разработка эволюционировала с течением времени, и даст возможность осознать, каким замечательным инструментом является ASP.NET. Опытные читатели могут пропустить разделы, которые покажутся им знакомыми.

HTML

HTML, или HyperText Markup Language (язык гипертекстовой разметки), представляет собой язык, который используется для вывода web-страниц. Он позволяет определять внешний вид и представление web-страницы: где именно должны располагаться графические элементы, каким образом должен выглядеть текст, а также все гиперссылки, имеющиеся на данной странице. Вообще говоря, любая web-страница, которую вы можете увидеть в Паутине, представляет собой не что иное как HTML-документ. Эти документы хранятся на web-сервере, и, когда клиент запрашивает какую-либо из них, сервер посылает ее клиенту посредством HTTP.

Существует большое количество отличных книг по программированию на HTML, поэтому мы не будем останавливаться на данной теме настолько подробно, насколько она того заслуживает. Для получения более детальной информации о том, как можно создавать прекрасные web-страницы посредством HTML, вы можете обратиться к книге "HTML 4.01 Programmers Reference" ("Руководство программиста по HTML 4.01), ISBN 1861005334, вышедшую в издательстве Wrox Press.

Теги

HTML состоит из последовательности тегов, которые описывают внешнее представление страницы. Эти теги заключаются в угловые скобки, для того чтобы браузер мог с легкостью идентифицировать их; теги используются браузером для корректного вывода и форматирования текста, который в них находится. Например, следующая строка выведет текст "Hello World", выделив его жирным шрифтом:

```
<b>Hello World</b>
```

Для подавляющего большинства тегов необходимо указывать открывающий и закрывающий теги. Это видно из примера, приведенного выше, в котором первый тег <b> "включает" выделение жирным шрифтом, а второй — "выключает" его для остального текста, который последует за данной строкой. Символ слэш '/' используется для обозначения закрывающего тега.

Базовый HTML

Единственным для HTML-страниц является правило, гласящее, что в них должен содержаться только один тег <html>. Таким образом, следующая строка представляет собой допустимую web-страницу:

```
<html></html>
```

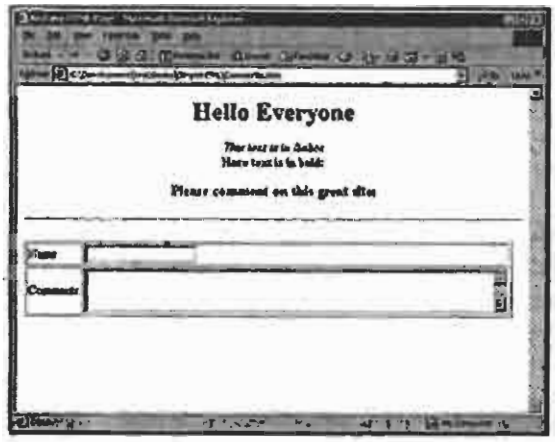
Очевидно, что такая страница совершенно бесполезна, поскольку в ней ничего не выводится. Все остальные теги должны располагаться между двумя этими HTML-тегами. Давайте создадим более полезную и интересную страницу, в которой будут использоваться некоторые базовые HTML-теги:

Практикум: простая HTML-страница

1. Откройте Notepad или какой-либо подходящий текстовый редактор, понимающий ASCII.
2. Ведите следующий текст в документ:

```
<html>
   <head>
      <title>An Easy HTML Page</title>
   </head>
   <body>
      <center>
         <h1>Hello Everyone</h1>
         <i>This text is in Italics</i><br>
         <b>Here text is in bold:</b>
         <h3><font color="Blue">
            Please comment on this great site:
         </font></h3>
         <hr>
         <br>
      </center>
      <table Border="1">
         <tr>
            <td>Name</td>
            <td><input type="text" name="Name"></td>
         </tr>
```

```
            <tr>
               <td>Comments:</td>
               <td>
                  <textarea name="Comments" rows=3 cols=65 wrap></textarea>
               </td>
            </tr>
         </table>
      </body>
   </html>
```



3. Сохраните файл под именем `Comments.htm`.
4. Запустите Internet Explorer или Netscape Navigator и откройте в нем только что сохраненный файл. Вы увидите вывод, аналогичный изображеиному на рисунке слева.

Как это работает — Простая HTML-страница

Хотя это совсем простая страница, она позволяет продемонстрировать основы программирования на HTML. Все содержимое файла заключено в блок `<html></html>`, который позволяет браузеру понять, что это HTML-файл.

В следующих строках кода показывается, каким образом можно задать название HTML-страницы. Тег `<title>` должен находиться внутри пары тегов `<head>`. Название страницы выводится в самой верхней части окна web-браузера. Область `<head>` может содержать и другую информацию относительно данной страницы, например, кто является ее создателем и каково ее содержание, и такая информация может быть использована различными поисковыми программами и web-агентами:

```
<head>
   <title>An Easy HTML Page</title>
</head>
```

Оставшаяся часть страницы заключается между двумя тегами `<body>`. Этот тег определяет содержимое реальной страницы, другими словами, то, что можно увидеть в клиентской области браузера:

```
<center>
   <h1>Hello Everyone</h1>
   <i>This text is in Italics</i><br>
   <b>Here text is in bold:</b>
```

Теги `<center>` используются, для того чтобы содержащиеся на странице элементы выравнивались относительно центра. Тег `<h1>` отиосится к стилю шрифта, который называется "heading level 1" (заголовок уровня 1). Существует шесть уровней заголовков (от 1 до 6), причем 1 соответствует самому большому. Стиль `<h1>` использован для форматирования названия странницы. В следующей строке используется тег `<i>`, который выделяет курсивом заключенный в нем текст наклонным. Тег `<b>` работает аналогичным образом, только заключенный в нем текст выделяется жирным шрифтом.

```
<h3><font color="blue">Please comment on this great site:</font></h3>
```

Далее мы используем тег <font> для указания цвета выводимого текста. Обратите внимание на то, что текст color="blue" содержится непосредственно внутри самого тега <font>. У многих тегов имеются дополнительные атрибуты, которые могут включаться в них для задания дополнительной информации. В данном случае атрибут color используется для задания цвета выводимого текста. В качестве значения этого атрибута может использоваться либо некоторое поименованное значение, например, blue, либо шестнадцатеричное число, которое представляет значения RGB выбранного цвета. Существует слишком много возможностей, использующихся в различных тегах, что не позволяет нам углубляться в подробности, но здесь приведен общий формат, которому должны следовать задаваемые атрибуты.

```
      <hr>
      <br>
   </center>
```

Следующая строка выводит горизонтальную линию через весь экран при помощи тега <hr>, что официально означает "horizontal rule" (горизонтальная линейка). Тег
 соответствует символу окончания строки. Этот символ завершает текущую строку и выводит следующий печатаемый символ со следующей строки. Обратите внимание на то, что ни тегу <hr>, ни тегу
 не требуется наличия закрывающего тега — они используются самостоятельно без второго совпадающего тега. Это следствие того, что ни один из них не предназначается для разметки выводимого текста — они просто выводят нечто на экран. После этого происходит закрытие тега, ответственного за выравнивание текста по центру, посредством закрывающего тега .

```
<table Border="1">
   <tr>
      <td>Name</td>
      <td><input type="text" name="Name"></td>
   </tr>
   <tr>
      <td>Comments:</td>
      <td><textarea name="Comments" rows=3 cols=65 wrap></textarea></td>
   </tr>
</table>
```

В следующих строках происходит создание простой таблицы, в строках которой содержатся текстовые окна. Остановимся на этом более подробно, для того чтобы продемонстрировать, каким образом можно создавать интерактивные HTML-страницы. Тег <table> сообщает браузеру о необходимости начать создавать таблицу. Между тегами <table> располагаются теги <tr> (table row — строка таблицы) и <td> (table data — табличные данные). Тег строки таблицы задает начало новой строки, а тег табличных данных — начало нового столбца. Существуют очень сложные приемы, которые могут использоваться для форматирования HTML-таблиц; автор советует познакомиться с ними подробнее, если заинтересовались web-разработкой.

Текстовые окна имеют внешний вид, хорошо знакомый вам по главе 13, где обсуждалась разработка приложений WebForm. Главное отличие заключается в том, что теперь для их вывода используется HTML, вместо того, чтобы создавать текстовые окна **TextBox** самостоятельно. Тег <input> обладает большим количеством атрибутов, которые могут использоваться совместно с ним, включая атрибут type, который определяет, является ли он текстовым окном, переключателем, командной кнопкой или окном с выбором. Тег <textarea> создает большее по размеру

текстовое окно с возможностью использования прокрутки, при этом в нем используются несколько атрибутов, определяющих различные свойства этого управляющего элемента,— такие как число строк, число столбцов, а также обрамляется текст или нет.

Как можно заметить, просмотрев данный файл в браузере, нами создана простая web-страница. Существует возможность изменять размеры окна, просматривать эту страницу в различных браузерах и вводить текст в два текстовых окна. Возможно, эта страница оставляет желать лучшего с точки зрения графического представления и содержания, однако основная идея заключалась в знакомстве с тем, каким образом используется HTML, а не в создании сногсшибательного web-сайта. Теперь мы приступаем к рассмотрению того, каким образом можно добиться от web-страницы передачи данных на сервер.

HTML-формы

Итак, у нас есть замечательная web-страница, однако как пользователь может передавать свои комментарии? На настоящий момент у него такой возможности нет — он может просто вводить текст в текстовые окна.

Необходим способ, который позволял бы отправлять комментарии читателя нашей страницы обратно к нам. Для реализации такого процесса в HTML используются формы. Можно представлять себе HTML-формы точно так же, как и обычные "бумажные" формы (бланки): в них содержатся поля, которые должны быть заполнены даннымн пользователем, после чего они отправляются надлежащему лицу или по надлежащему адресу.

Форма создается точно так же, что и большинство HTML-элементов — с помощью тегов. Тег `<form>` используется для размещения формы на странице, а тег `</form>` завершает ее создание. Этот тег ничего не выводит на экран, он просто обозначает начало и конец формы, а поля, расположенные между ними, позволяют задавать фактические поля данной формы.

Есть несколько очень важных атрибутов, которым необходимо присваивать соответствующие значения в теге `<form>`. Основной синтаксис при этом имеет следующий вид:

```
<form name="myform" method="post" action="myform.asp">
```

Атрибут `name` идентнфицирует форму и должен быть уникальным, иначе сервер не сможет корректно обработать форму. Атрибут `method` определяет, каким способом данные из формы будут переправляться серверу. Наконец, атрибут `action` указывает браузеру, куда ему следует отправлять данные, введенные в форму.

Атрибут `method` может обладать одним из двух значений: `GET` или `POST`. Это должно быть знакомо по предшествующему обсуждению нами HTTP-запросов. В форме указывается, в каком формате браузер должен отправить запрос серверу после ввода в форму данных. Помните, что метод `GET` будет присоединять введенные данные к адресу ресурса, создав строку запроса, которая и будет анализироваться сервером. Метод `POST` будет отправлять введенные в форму данные непосредственно в внде тела HTTP-сообщення. Метод `GET` оказывается очень удобным при работе с небольшим количеством текстовых данных, в то время как метод `POST` может использоваться с произвольным количеством данных любого формата.

И последняя ключевая информация, которая необходима для создания формы,— это какие действия будет выполнять данная форма на сервере после ее заполнения. Это задается с помощью атрибута `action` тега `<form>`. В большинстве случаев он будет указывать на активную серверную web-страннцу, которая может обработать

введенные данные и выдать пользователю какой-либо ответ. В следующем разделе мы рассмотрим пересылку введенных в форму данных непосредственно активной странице, но на настоящий момент мы ограничиваемся простой отправкой данных по электронной почте.

В прошлом единственным способом создать действительно динамичное содержание страницы можно было только посредством использования Common Gateway Interface (CGI — интерфейс общего шлюза), который позволял web-страницам вызывать приложения, расположенные на web-серверах. Такие приложения могли создаваться на произвольном языке программирования, но чаще всего они писались на C/C++ или на Perl. В этом случае форма передавалась приложению, которое возвращало обратно получившееся динамическое содержимое. Однако с точки зрения сервера такие CGI-приложения не были дружественными по отношению к процессору или ресурсу, поэтому во многих случаях они были заменены на инструменты создания сценариев, работающих на стороне сервера, с которыми мы познакомимся в следующем разделе. Во многих web-приложениях HTML-форма передавалась динамической странице, находящейся на стороне сервера, которая выполняла обработку введенных пользователем данных и генерировала соответствующую ответную страницу.

Местонахождение приложения или страницы, которым будет передаваться форма, задается атрибутом `action` тега `<form>`. Он может ссылаться на CGI-приложение, активную web-страницу и даже на адрес электронной почты. В последнем случае заполненная форма передается по электронной почте по указанному адресу.

Давайте создадим небольшую программу, которая позволит увидеть, как эти формы работают на практике.

Практикум: HTML-формы

1. Откройте файл `Comments.htm`, созданный в предыдущем примере.

2. Добавьте в существующий документ следующие строки:

```
<html>
   <head>
      <title>An Easy HTML Page</title>
   </head>
   <body>
      <center>
         <h1>Hello Everyone</h1>
         <i>This text is in Italics</i><br>
         <b>Here text is in bold:</b>
         <h3><font color=Blue>Please comment on this great site:</font></h3>
         <hr>
         <br>
      </center>
      <form method="post" enctype="text/plain"
         action="mailto:yourname@domain.com" name="CommentsForm">
         <table border="1">
            <tr>
               <td>Name</td>
               <td><input type="text" name="Name"></td>
            </tr>
            <tr>
               <td>Comments:</td>
               <td>
                  <textarea name="Comments" rows=3 cols=65 wrap></textarea>
               </td>
```

```
            </tr>
          </table>
        <br>
      <input type="submit" value="Submit">
      </form>
      </body>
    </html>
```



3. Введите ваш адрес электронной почты в атрибут `action` сразу за текстом `mailto`.
4. Сохраните документ под именем `FormComments.htm`.
5. Откройте этот HTML-файл в браузере. Введите какие-нибудь данные и нажмите кнопку Submit (см. рис. слева).
6. На экране при этом ничего не изменится, но если ваш браузер правильно настроен, то вы получите сообщение по электронной почте. В зависимости от используемого браузера передаваемые данные будут содержаться либо в приложении, либо непосредственно в самом теле сообщения. Но в любом случае полученные данные будут выглядеть следующим образом:

```
Name=Keith
Comments=This site is amazing!
```

Как это работает — HTML-формы

К этому примеру добавлены всего три строки, но при этом его возможности изменились коренным образом. Теперь есть способ, позволяющий конечным пользователям, просматривающим нашу страницу, использовать обратную связь. Это первый шаг на пути к созданию по-настоящему интерактивных web-страниц.

В следующей строке начинается описание формы, и в ней также определяются действия, которые эта форма будет выполнять, когда пользователь закончит ее заполнение:

```
<form method="post" enctype="text/plain"
      action="mailto:yourname@domain.com" name="CommentsForm">
```

Атрибут `method` указывает на то, что данная форма будет отправлять на сервер команду HTTP POST, содержащую введенные в форму данные в теле HTTP-сообщения. Атрибут `action` определяет, куда будет происходить отправка данных. В данном случае форма будет отправлять данные по электронной почте по нашему адресу. Атрибут `enctype` определяет тип данных, отправляемых в команде HTTP POST. В большинстве случаев это простой текст. Атрибут `name` используется для того, чтобы можно было ссылаться на элементы формы из языка JavaScript, с чем мы столкнемся несколько позже.

Между тегами `<form>` мы определяем структуру таблицы, кроме того, там же располагаются теги `<input>` и `<textarea>`. Обратите внимание на то, что для обоих тегов весьма важным является атрибут `name`, который задается в исходной форме. Отправка данных из формы на сервер происходит в виде последовательности пар имя/значение, в которых в качестве имени используется значение атрибута `name` соответствующего поля формы, а в качестве значения — данные, введенные пользователем. Следовательно, единственный способ получить доступ к конкретным

элементам данных заключается в использовании имен соответствующих элементов формы. То, как это работает, можно увидеть ниже по данным, полученным по электронной почте:

```
Name=Keith
Comments=This site is amazing!
```

Для создания кнопки Submit снова используем тег `<input>`, однако на этот раз атрибуту `type` присваивается значение `submit`. Это приводит к тому, что форма, в которую включен этот тег, выполняет атрибут `action`, предполагающий отправку электроиной почты. Атрибут `value` определяет, какой текст должен выводиться на самой кнопке,— точно так же, как мы это делали с кнопками WinForm.

Элементы формы

Существует перечень форм, которые можно размещать на web-странице. Для базового HTML выбор несколько ограничен по сравнению с возможностями, имеющимися в ASP.NET (см. главу 24). Следующая таблица иллюстрирует различные возможности форм, которыми располагает базовый HTML:

Элемент формы	HTML-тег	Примечания
Текстовое окно	`<input type="text">`	Этот элемент позволяет создавать такое же текстовое окно, что в приложениях WinForm. Пользователь может вводить текст в определенную область, а также использовать стандартные операции вырезания и вставки.
Текстовое окно для пароля	`<input type="password">`	Текст, вводимый в такое окно, выводится в виде последовательности скрытых символов. Обратите внимание на то, что это не означает, что данный текст скрывается при передаче; для безопасной передачи следует воспользоваться какими-либо другими средствами.
Многострочное текстовое окно	`<textarea>`	Этой особой разновидностью текстового окна следует пользоваться в тех случаях, когда необходимо работать более чем с одной строкой текста. Этот управляющий элемент позволяет создать несколько строк текста, при этом предоставляются стандартные возможности их редактирования.
Переключатели	`<input type="radio">`	Переключатели используются, когда необходимо предоставить пользователю сделать выбор из нескольких возможностей, причем допускается выбор только одного из имеющихся вариантов. В каждый конкретный момент времени пользователь может выбрать только один переключатель.
Окна с выбором	`<input type="checkbox">`	Окна с выбором работают аналогично переключателям за исключением того, что они позволяют пользователю одновременно выбирать несколько значений из одной группы. Таким образом, окна с выбором являются естественными решением, когда требуется ответить на вопрос типа 'укажите все подходящие варианты'.
Меню	`<select name="MenuName">`	Этот управляющий элемент позволяет создавать меню, которое дает пользователю возможность осуществлять выбор из множества опций. Каждый пункт меню задается с помощью отдельного тега `<option>`.

Продолжение таблицы

Элемент формы	HTML-тег	Примечания
Скрытые поля	`<input type="hidden">`	Скрытые поля никогда не выводятся пользователю, и их удобно использовать для хранения данных, которые подлежат обработке на сервере. В следующей главе вы познакомитесь с тем, каким образом ASP.NET использует скрытые поля для хранения информации о текущем сеансе.
Образ	`<input type="image">`	Этот управляющий элемент позволяет создавать образ, щелкнув на котором мышью, пользователь может передавать введенные в форму данные. Помимо других заданных в форме данных серверу отправляются горизонтальная и вертикальная координаты курсора мыши. Это позволяет серверу возвращать содержимое в зависимости от того, в каком месте образа пользователь щелкнул кнопкой мыши. В качестве широко распространенного на многих web-страницах примера можно привести карту расположения различных подразделений компании. Щелкнув мышью на названии города или штата, пользователь получает возможность перейти на страницу, где находится информация о соответствующем филиале компании.
Кнопка Submit	`<input type="submit">`	По умолчанию этот управляющий элемент выводится в виде кнопки, хотя существует возможность изменять его внешний вид. При нажатии на эту кнопку происходит передача формы со всеми имеющимися данными серверу.
Кнопка Reset	`<input type="reset">`	Этот управляющий элемент позволяет отказаться от всех выбранных пользователем возможностей и удалить введенный текст из всех управляющих элементов так, как будто пользователь только что зашел на данную страницу.

Неизменяемые web-страницы в сопоставлении с активными

Все страницы, рассмотренные до настоящего момента, являются неизменяемыми web-страницами. Это означает, что каждая из них представляет собой HTML-файл, расположенный на сервере, и что можно изменить такие страницы, только изменив HTML-файлы. Когда World Wide Web только появилась, большинство сайтов состояло из нескольких взаимосвязанных неизменяемых страниц. При необходимости внести в них какие-либо изменения web-разработчику приходилось просматривать HTML-файл и вносить в него соответствующие изменения. Объем работы, требовавшийся для сопровождения сайтов, у которых имелись постоянно изменяющиеся элементы и новостные сообщения, сделало практически невозможным использование в них неизменяемых web-страниц.

Возникла насущная необходимость в способе, который бы позволял создавать web-страницы "на лету", без использования разработчика каждый раз, когда возникала необходимость внесения в страницу каких-либо изменений. Добавляется какой-либо новый элемент в системе базы данных на серверной части — он автоматически появляется на web-сайте. Появляются какие-либо новостные сообщения — они появляются на экране в режиме онлайн. Кроме того, использование

HTML в качестве клиентской части системы работы с базой данных позволяло создавать универсальные системы. Любой, кто обладает некоторым браузером, имест возможность получить доступ к вашему приложению безотносительно операционной системы, типа компьютера и т. п. Все, что в таком случае необходимо клиенту,— это web-браузер, и следовательно, для осуществления доступа к данным нет никакой необходимости инсталлировать клиентское приложение.

Вскоре после того, как World Wide Web стала завоевывать популярность, появилась понимание важности активных серверных страниц и начались работы над различными решениями этой проблемы. Существует два способа предоставления пользователю более динамичного режима работы — создание клиентских и серверных сценариев.

Создание сценариев на стороне клиента

Чтобы web-страницы имели возможность работать в более динамичном и интерактивном режиме, в них можно поместить блоки кода, которые содержатся в самой HTML-странице. Такой код не подлежит компиляции, подобно программам, с которыми мы имели дело на протяжении всей этой книги. Программа, написанная на языке описания сценария, интерпретируется по мере своего выполнения. Обладая кодом, который может взаимодействовать непосредственно с самой web-страницей и выполняться на ней, программист получает более широкие возможности, чем в случае неизменяемой web-странницы. Становится возможным выполнять на клиентском компьютере весьма интересные вещи без обращения к серверу. Это может оказаться чрезвычайно полезным для проверки допустимостн содержнмого управляющих элементов HTML на web-странице, выполняемой перед отправкой формы на сервер, или для создания пользовательского интерфейса, который бы предоставлял пользователям гораздо более широкие возможности.

Для большинства web-прнложений в качестве языка описания сценариев используется JavaScript. Названный язык работает на всех коммерческих браузерах, в нем имеется большое количество команд и объектов. В настоящей книге мы не собираемся подробно останавливаться на языке JavaScript; если вы хотите узнать дополнительную информацию, то можете воспользоваться также вышедшей в издательстве Wrox Press книгой *Beginninig JavaScript* (Введение в JavaScript), ISBN 1681004060.

Вернемся к последней web-странице и включим в нее некоторые проверки допустимости введенной информации, которые будут выполняться перед отправкой формы на сервер.

Практикум: Создание сценариев на стороне клиента на языке JavaScript

1. Откройте файл `FormComments.htm` из предыдущего примера.
2. Добавьте приведенный ниже код в верхнюю часть страницы. Мы собнраемся добавить некоторую функцию на JavaScript, которая будет проверять допустимость нформацин в различных элементах формы на страннце:

```
<html>
   <head>
      <title>An Easy HTML Page</title>
      <script language="JavaScript">
```

```
<!--

function ValidateForm()
{
   if (document.CommentsForm.Name.value=="")
   {
      alert("Please enter a name!");
      document.CommentsForm.Name.focus();
      return false;
   }

   if (document.CommentsForm.eMail.value=="" ||
      document.CommentsForm.eMail.value.indexOf("@") == -1)
   {
      alert("Please enter a valid eMail address!");
      document.CommentsForm.eMail.focus();
      return false;
   }

   if (document.CommentsForm.Comments.value=="")
   {
      alert("Please enter some comments!");
      document.CommentsForm.Comments.focus();
      return false;
   }

   return true;
}

// -->
</script>
</head>
```

3. Изменим следующую строку, чтобы включить ссылку на функцию JavaScript при передаче формы:

```
<form method="post" enctype="text/plain"
   action="mailto:yourname@domain.com" name="CommentsForm"
   onSubmit="return ValidateForm()">
```

4. Добавим управляющий элемент формы, отвечающий за электроинную почту. Добавьте следующие строки для включения нового текстового окна для электронной почты:

```
<table border=1>
   <tr>
      <td>Name</td>
      <td><input type="text" name="Name"></td>
   </tr>
   <tr>
      <td>Email</td>
      <td><input type="text" name="eMail"></td>
   </tr>
   <tr>
      <td>Comments:</td>
      <td><textarea name="Comments" rows=3 cols=65 wrap></textarea></td>
   </tr>
</table>
```

5. Сохраните страницу под именем `JavaScriptForm.htm`.



6. Загрузите эту страницу в свой браузер. Поэкспериментируйте над ней, вводя недопустимые значения и наблюдая окна с сообщениями, которые будут открываться. До тех пор пока все три окна не будут заполнены допустимыми данными, форма не будет переслана (см. рис. слева).

Как это работает — Создание сценариев на стороне клиента на языке JavaScript

JavaScript используется для выполнения заданий на компьютере клиента. Среди этих задач может быть все, что угодно, начиная от проверки допустимости, осуществляемой в настоящем примере, и заканчивая выполнением различных манипуляций над элементами страницы и созданием более динамичного интерфейса пользователя. С помощью языка JavaScript можно реализовать весьма интересные вещи, но поскольку не это является основной темой данной главы, мы остановимся на простом примере использования этого языка. Основное преимущество языка JavaScript заключается в том, что он позволяет избежать цикла обращение к web-серверу — получение ответа от web-сервера, что оборачивается экономией времени и ширины полосы пропускания, и, кроме того, все вычисления выполняются на клиентском компьютере вместо использования дорогостоящих ресурсов сервера.

В рассматриваемом примере мы добавили в форму некоторую простую проверку допустимости. JavaScript — это язык программирования, синтаксис которого напоминает синтаксис C#, поскольку оба этих языка произошли от языка C, и большая часть кода является абсолютно понятной. Мы создаем функцию с именем `ValidateForm()`. Эта функция осуществляет проверку всех трех текстовых окон на допустимость содержащихся в них данных. Обратите внимание на то, что эта функция обрамляется тегами `<script>`. Это делается для того, чтобы браузер мог начать интерпретировать последующий текст как команды JavaScript. Сам язык программирования задается с помощью атрибута `language`. Кроме того, все команды JavaScript заключаются в блок комментариев HTML между символами `<!--` и `// -->`. Это позволяет гарантировать, что браузеры более ранних версий, которые не работают с JavaScript, проигнорируют все эти команды.

```
<script language="JavaScript">
<!--
```

Сама функция выполняет проверку на допустимость данных в трех элементах формы. JavaScript обращается ко всему, что находится в браузере, включая собственно HTML-страницу, с помощью **Document Object Model (DOM, объектная модель документа)**. DOM задает иерархическую систему обращения ко всем элементам внутри страницы и позволяет выполнять над ними различные манипуляции, получать их значения и обновлять их:

```
if (document.CommentsForm.Name.value=="")
{
   alert("Please enter a name!");
   document.CommentsForm.Name.focus();
   return false;
}
```

Оператор `if` использует DOM для доступа к конкретному элементу формы. В нем указывается имя формы — `CommentsForm` и имя элемента формы — `Name`, что позволяет сужать область поиска до тех пор, пока в ней не останется единственный элемент. Затем в операторе используется какое-либо свойство этого элемента — в данном случае `value`, для получения доступа к хранящемуся в нем тексту. Далее этот текст сравнивается с пустой строкой, и если результатом этого сравнения является значение `true`, то выдается сообщение об ошибке и устанавливается фокус с помощью метода объекта данного элемента формы `focus()`.

Остальные сравнения выполняются аналогичным образом. Проверка допустимости адреса электронной почты оказывается несколько сложнее, поскольку, кроме проверки того, что данное окно не является пустым, необходимо также убедиться в наличии символа @. Для реализации этого используется метод `IndexOf()`. Этот метод возвращает либо позицию переданного ему символа, либо — 1, если ничего не найдено. Далее выполняется проверка равенства возвращенного значения с –1 и, если это так, то выводится сообщение об ошибке.

Эта функция вызывается, когда делается попытка передать форму путем изменения тега `<form>`

```
<form method="post" enctype="text/plain"
action="mailto:yourname@domain.com" name="CommentsForm"
onSubmit="return ValidateForm()">
```

Когда форма отправляется, выполняется действие, представленное в `onSubmit`. В данном случае мы вызываем функцию `ValidateForm()`, которая только что была описана. Если `ValidateForm()` возвращает значение `false`, то форма не будет пересылаться серверу — именно таким способом можно предотвратить отправку формы, если в процессе проверки обнаруживаются ошибки.

Большинство объектов формы обладают событиями, подобными `onSubmit`, которые могут быть перехвачены, и запустить соответствующий код на языке JavaScript. Некоторые из наиболее полезных событий перечислены ниже, хотя не все они могут быть доступными для произвольного объекта:

Имя события	Описание
`onFocus`	Исполняется, когда браузер загружается в первый раз и когда пользователь переключается на браузер в Windows. Это же событие генерируется, если пользователь переводит фокус на данный управляющий элемент.
`onLoad`	Наступает после того, как все рисунки и весь текст для данной страницы, загружены. Это событие может быть использовано для инициализации любых объектов, расположенных на странице.
`onResize`	Возникает, когда пользователь изменяет размеры окна в браузере.
`onUnload`	Возникает, когда пользователь переходит на новую страницу или когда динамический сценарий начинает пересылать пользователю новое содержимое текущей страницы.
`onBlur`	Наступает, когда пользователь покидает некоторый управляющий элемент.
`onClick`	Это событие возникает, когда пользователь щелкает кнопкой мыши.

Создание сценариев на стороне сервера

Использование сценариев на стороне клиента не решает основной проблемы — создания web-сайтов, которые действительно управляются данными. Хотя клиентские сценарии оказываются весьма полезными, они используются исключительно для решения некоторых частных задач; по-настоящему же необходимым инструментом является создание сценариев на стороне сервера.

Серверный сценарий означает, что существует код, который при посылке пользователем запроса выполняется, помогая создать запрашиваемую страницу. Этот код может извлекать данные из базы данных, строить пользовательский интерфейс в зависимости от полученных данных, а затем отправлять HTML-страницу обратно клиенту. Конечным результатом такого процесса будет отправка браузеру стандартной web-страницы на HTML. Этот процесс представляет собой динамическое создание web-страниц — web-страница, показываемая в браузере, на самом деле является продуктом работы кода, который был запущен после клиентского запроса; в действительности такой страницы физически не существует.

Другим огромным преимуществом серверного сценария является снижение временных затрат на сопровождение приложений разработчиками. Вместо того, чтобы устанавливать приложение Windows на каждое рабочее место в компании, можно создать одно web-приложение, к которому имеется общий доступ. Если возникает необходимость внесения в это приложение каких-нибудь изменений или исправлений, то достаточно обновить приложение только на сервере, а не на всех клиентских компьютерах. Это также позволяет каждому клиенту постоянно пользоваться самой последней версией приложения, поскольку именно она всегда будет находиться на сервере.

Существует большое количество различных инструментов и языков программирования, которые позволяют создавать активные web-страницы. Вот некоторые из наиболее популярных технологий:

- Active Server Pages
- PHP
- ColdFusion
- WebSphere
- Java Server Pages

Настоящая книга посвящена разработкам в среде .NET, поэтому мы посвятим некоторое время решению этой проблемы компанией Microsoft, а именно Active Server Pages. В следующей главе будет подробно рассмотрена система Active Server Pages последнего поколения — ASP.NET.

Active Server Pages

Active Server Pages — это технология компании Microsoft, предназначенная для создания динамических web-страниц. В ней используется язык программирования VBScript, который представляет собой подмножество языка Visual Basic. Он работает очень похоже на Visual Basic за исключением того, что все переменные в нем принудительно имеют вариантный тип и исключены некоторые функциональные возможности, касающиеся например, ввода/вывода. Поскольку это — язык описания сценариев, то подразумевается отсутствие какой-либо компиляции; весь код считывается по мере выполнения его сервером, что дает существенный выигрыш во времени по сравнению с компилируемыми языками программирования.

Это очень большая тема, а мы планируем в следующей главе подробно остановиться на ASP.NET, поэтому эту технологию мы не будем изучать особенно тщательно. Если вы захотите ознакомится с ней более детально, то можете обратиться к вышедшей в издательстве "Лори" книге "Введение в Active Server Pages 3.0" (*Beginning Active Server Pages 3.0*) в 2001 г.

Управление состоянием

Управление состоянием в web-приложении — это очень сложная вещь. Состояние — это термин, используемый для описания постоянной информации, которая должна сохраняться не только на протяжении отдельного обращения или посещения web-сайта. Поскольку HTTP — это протокол без сохранения состояния, то web-приложений не могут определить, какой именно пользователь посещает их сайт, а также то, посещает ли он данный сайт в первый раз или вернулся на него.

Одним из способов решения этой проблемы являются куки (cookie) — технология, позволяющая сохранять информацию на клиентской машине. Это может быть произвольная информация — ключ просмотра БД, какой-либо интерес пользователя или какая-либо другая информация, которая может использоваться сайтом для отслеживания своих пользователей. Когда некоторый пользователь обращается к сайту, то сервер посылает соответствующие куки клиентской машине, где оно и хранится. Когда этот же пользователь посещает сайт повторно, браузер посылает серверу предварительно сохраненные куки, позволяя сайту либо персонифицировать свое содержимое, либо просмотреть реальную потребительскую корзину этого покупателя, либо вообще выполнить какие-то действия. Теперь у web-сайта появляется возможность сохранять свое состояние между последовательными посещениями различных пользователей. Другой способ отслеживать пользователей, посещающих ваш сайт,— хранить записи об этих посещениях в базе данных. Этот подход оказывается очень полезным, если вам требуется реализовать проверку прав доступа на стороне сервера, при которой идентификатор и пароль пользователя могли бы сравниваться со значениями, хранящимся в базе данных самого пользователя.

Эволюция Паутины

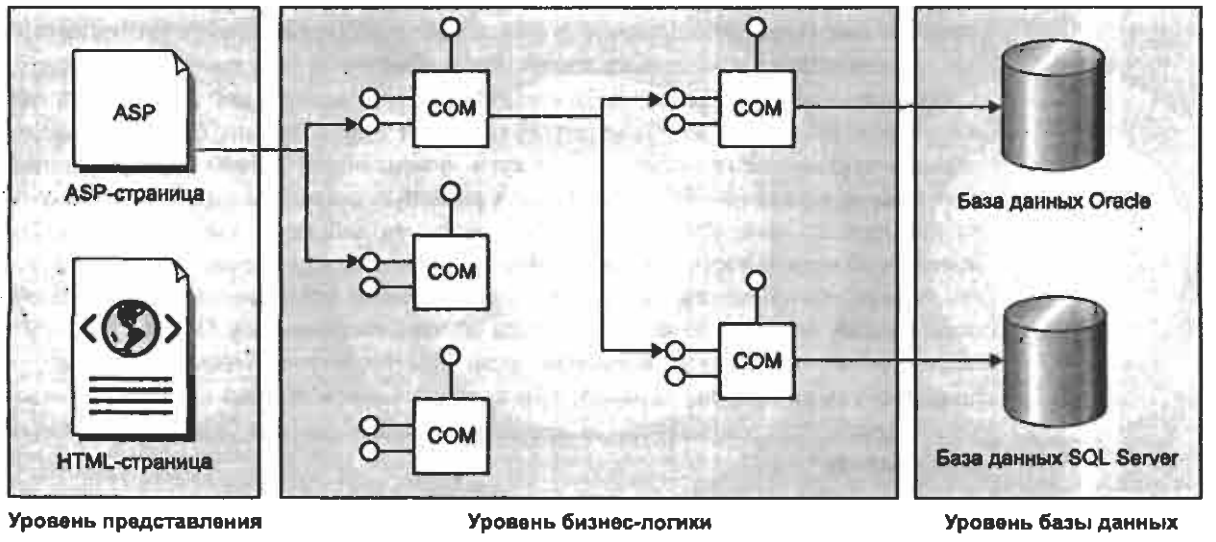
Появление web-страниц ASP было знаменательным событием. Они позволяли решить фундаментальную проблему функционирования World Wide Web. Возможность создавать активные web-страницы на основе информации, извлекаемой из баз данных или других систем, расположенных на стороне сервера, была поистине революционной. Большинство сайтов продолжают использовать эту технологию и сегодня.

Многие обратили внимание на то, что при выполнении сценариев на стороне сервера существуют некоторое фундаментальное ограничение. Поскольку такие страницы не компилируются, а создаются в динамике по мере поступления запроса от клиента, то время ожидания ответа может оказаться достаточно большим. А поскольку на создание каждой страницы требуется достаточно много времени, сервер может оказаться не в состоянии справиться с нагрузкой, если к нему будет поступать тысячи обращений в секунду.

По этой причине была разработана другая архитектура, в которой бизнес-логика и доступ к данным были разделены на независимые компоненты, которые выполнялись в виде откомпилированных COM-компонентов. Эта разработка получила название трехзвенной, поскольку приложение расщеплялось на три разные части: уровень клиента/представления, промежуточные COM-компоненты и хранилище данных на стороне сервера. Эта модель очень быстро превратилась в многозвенную модель, поскольку промежуточное звено начало разрастаться и включать в себя различные уровни COM-компонентов и приложений.

После выделения различных уровней появилась возможность независимой разработки каждого из них, поскольку интерфейсы между уровнями оставались неизменными. Это повлекло за собой разработку компонентов, которые в минимальной степени основывались бы на работе со сценариями Active Server Pages. ASP по-прежнему оставались необходимыми для вызова этих компонентов и для форматирования возвращаемых ими данных, поскольку они представляли собой уровень пользовательского интерфейса для web-приложений.

Такая модель оказалась чрезвычайно эффективной для создания масштабируемых, легкодоступных web-приложений уровня отдельного предприятия:



Web-программирование в .NET

Система .NET представляет собой очередной революционный шаг в разработке web-приложений уровня предприятия. С учетом того, что в ней особое внимание уделяется компонентам и совместному использованию, она представляет собой среду программирования следующего поколения. ASP.NET — это система .NET в приложении к активным серверным страницам. При помощи ASP.NET появляется возможность создавать компилируемые управляемые данными web-страницы, у которых есть полноценный доступ ко всем аспектам .NET Framework и к тем функциональным возможностям, которыми она обладает.

Для web-разработчика это означает, что ASP больше не является самым слабым звеном в цепочке разработки многозвенного приложения. Теперь есть возможность разрабатывать действительно понятные компилируемые web-приложения, которые возвращают клиенту непосредственно HTML, и при этом вести разработку в среде, предоставляющей для таких разработок богатые возможности.

ASP.NET

В отличие от ASP, в ASP.NET не используется собственный язык программирования. Напротив, приложения ASP.NET создаются точно так же, как приложения, которые мы создавали на протяжении всей книги,— на C#, на VB.NET или на любом другом языке программирования, который совместим с .NET Framework.

Это позволяет web-программисту отделять содержимое от его представления, поскольку приложение ASP.NET может использовать WebForms для вывода пользовательского интерфейса, а также применять весь имеющийся в .NET Framework инструментарий, с которым мы знакомились в этой книге, для создания программируемой серверной части.

Код, из которого состоит приложение ASP.NET, представляет собой компилируемую программу .NET, следовательно, он будет выполняться значительно быстрее, чем аналогичное ASP-приложение. Только одна эта особенность может привести к существенному увеличению быстродействия. Добавьте к этому преимущества модели автоматического кэширования и простоту использования системы управления состоянием, и вы поймете, какая это замечательная среда разработки.

Ниже приводится краткий перечень некоторых преимуществ ASP.NET:

- Компилируемый код, допускающий раннее связывание, строгую типизацию и своевременную (Just-In-Time, JIT) компиляцию в родной код
- Файлы конфигурации на основе XML
- Полная совместимость с .NET Framework
- Простота использования модели кэширования
- Полностью автоматизированное управление состоянием

WebForms

WebForms — это Windows Forms для World Wide Web. Точно так же, как мы создавали приложение Windows Forms в главе 13, можно создать интерактивные, обладающие широкими возможностями web-страницы с использованием большей части того инструментария, который использовался нами для быстрого создания приложения Windows Forms.

Такие web-страницы обладают многими преимуществами по сравнению со стандартными ASP-страницами. Одно из таких преимуществ состоит в том, что программисту не приходится беспокоиться о проблемах, связанных с зависимостью от конкретного браузера. ASP.NET оттранслирует WebForms в версию HTML, не зависимую от браузера и нейтральную по отношению к платформе. Кроме этого, у web-разработчика есть доступ к разнообразному набору событий и свойств, что во многом напоминает программирование в стандартной среде Windows. Эти события могут обрабатываться как на стороне клиента, так и на стороне сервера и, кроме того, они позволяют использовать интуитивный подход к web-разработке, при котором не приходится беспокоиться по поводу сложных аспектов обработки и интерпретации HTML-форм. Помимо этого, управляющие элементы .NET могут быть использованы для создания интерфейса, стандартного с точки зрения пользователя, а ASP.NET позволяет разработчику определять события и свойства этих управляющих элементов. Все это, в конечном итоге, приводит к тому, что в распоряжении разработчика оказывается среда с полным набором возможностей, позволяющая создавать эффективные, масштабируемые и рациональные web-приложения. Ниже перечислены основные преимущества WebForms:

- Может выполняться на произвольном браузере и представляет корректную для данного браузера версию HTML
- Допускает использование любого языка программирования, совместимого с .NET

- Обеспечивает полную поддержку инструментов разработки GUI
- Предоставляет широкий выбор клиентских и серверных управляющих элементов
- Позволяет легко разделять код, ответственный за представление от кода, реализующего бизнес-логику

Итоги

World Wide Web прошла долгий путь развития и теперь является стройной средой с богатыми возможностями. ASP.NET — это инструмент, предназначенный для высвобождения всей мощи .NET Framework и использования ее для создания отличных web-сайтов.

В настоящей главе мы познакомились с историей Интернета и World Wide Web, а также с тем, каким образом разработчики могли создавать замечательные web-сайты на протяжении всей истории Паутины. Мы изучили основы HTTP и HTML, а также узнали о множестве причин, по которым оказывается необходимым использование сценариев как со стороны клиента, так и со стороны сервера. Мы также разобрали пример использования JavaScript для выполнения на стороне клиента проверки допустимости данных и узнали, какую роль играют Active Server Pages.

Кроме этого, мы обсудили, почему ASP.NET является шагом вперед в области web-разработки и какие преимущества дает использование ASP.NET и WebForms. В следующей главе мы будем изучать ASP.NET более детально и создадим работающий пример.



Приложения ASP.NET

ASP.NET — это способ, предусмотренный в .NET для создания динамических web-сайтов. Как и ADO.NET, ASP.NET заимствует свое название от предшествующей технологии Active Server Pages (ASP — активные серверные страницы), но в данном случае сходство этим практически и ограничивается. В то время как ASP-страницы интерпретировались и создавались на функционально ограниченных языках описания сценариев VBScript и JScript, приложения ASP.NET компилируются на язык MSIL точно также, как и другие приложения .NET. Это означает, что у нас появляется возможность описывать их на тех же языках программирования, что и остальные приложения .NET. В данном случае особенно важно (поскольку книга посвящена языку программирования C#), что можно писать приложения ASP.NET на C#.

Другим огромным преимуществом ASP.NET является использование управляющих элементов на стороне сервера. В ASP также имелась возможность включать управляющие элементы HTML (такие, как кнопки и другие элементы форм) в создаваемые страницы, однако делать это приходилось вручную, а реагировать на любой ввод пользователя в такие элементы (например, пользователь щелкнул мышью на кнопке) можно было либо с помощью выполнения сценария JavaScript на клиентской стороне, либо повторно отправляя страницу серверу. В первом случае проблема заключается в том, что в различных браузерах используются разные реализации JavaScript, поэтому практически не представляется возможным создавать такие изощренные программы, которые могли бы работать как на Netscape, так и IE. Кроме того, существуют пока такие допотопные браузеры, которые вообще не поддерживают JavaScript. Интернет не сохраняет свое состояние, поэтому при повторной отправке страницы на сервер теряется вся информация, хранящаяся в переменных страницы. Чтобы этого не произошло, следует создать довольно сложный код, который позволит сохранять эти переменные в HTML-элементах или в строке запроса URL.

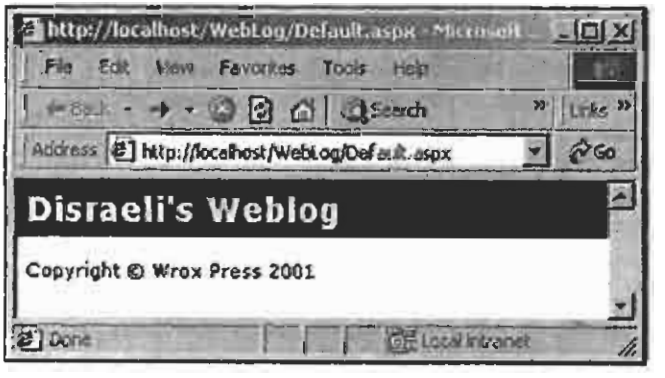
ASP.NET позволяет разрешить эти проблемы за счет использования управляющих элементов *на стороне сервера*. Такие элементы генерируют HTML-код, который передается браузеру для вывода управляющего элемента, но они, кроме того, генерируют еще и функции на JavaScript и скрытые управляющие элементы, в которых сохраняется их текущее состояние. Если страница передается серверу, то вместе с ней серверу передается и вся соответствующая информация; управляющий элемент автоматически обрабатывает эту информацию и изменяет HTML таким образом, чтобы вывести управляющий элемент на экран. В простейшем варианте это означает, что мы получаем возможность использовать такие текстовые окна, которые "запоминают" значения, введенные пользователем, но если посмотреть немного дальше, то они позволяют ASP.NET использовать более сложные управляющие элементы (например, управляющий элемент календарь), чем те, с которыми мог справиться ASP.

Последнее, и, возможно, наиболее важное преимущество заключается в том, что ASP.NET предоставляет простой способ создания **web-служб**. На текущий момент можно представить себе web-службы просто как составные части Паутины, однако они являются гораздо более сложной темой, и им будет посвящена следующая глава целиком.

Для того чтобы увидеть какой-либо код, работающий на стороне сервера, в действии, зайдите на сайт Amazon.com и выберите свой любимый роман. Страницы с подробной информацией о романе, которую вы видите перед собой, не существует на сервере. На самом деле на сервере имеется некоторый шаблон, в котором описывается общий вид страницы, и, когда вы запрашиваете какую-нибудь страницу, этот код выполняется на сервере, извлекает информацию о затребованной книге из базы данных и вставляет эту информацию в шаблон. То, что возвращается в ответ на ваш запрос,— это HTML; мы поговорим об этом несколько позже.

На протяжении всей главы мы будем заниматься кодом, который выполняется на стороне сервера. Вашему браузеру придется выполнять относительно небольшой объем работы при выводе страницы.

Начиная с этого момента, под "активными страницами" будут иметься в виду страницы, которые могут быть созданы посредством как классического ASP, так и ASP.NET. Обе технологии используют один и тот же основной подход к созданию страниц — просто ASP.NET является намного более мощным средством, и вы сами поймете, насколько существенно упрощается создание web-сайтов при использовании ASP.NET.



Рассмотрим пример. Он выглядит как обыкновенная web-страница (см. рис. слева).

Обратите внимание на строку с информацией об авторских правах в нижней части страницы. Представим себе, что эта строка появляется на каждой странице нашего web-сайта. Сейчас она относится к 2001 году, и это означает, что при наступлении нового, 2002 года нам придется пройти по всему сайту и для каждой страницы изменить значение 2001 на 2001–2002.

При использовании "активных страниц" можно сконфигурировать страницу таким образом, что она будет самостоятельно определять год и вносить изменения автоматически. Отсюда следует, что мы освобождаемся от решения этой простой, но очень нудной административной задачи на все последующие годы.

Это очень банальный пример. Технология создания активных страниц вступает в свои права, когда вы привязываете свой сайт к какой-нибудь базе данных или к каким-либо другим функциональным возможностям, подобным тележке покупателя в электронной коммерции. В этой главе мы планируем рассмотреть, каким образом можно создать "web-регистратор". Он является web-аналогом обычного дневника, который мы, вместо того, чтобы записывать его на бумагу и запирать в ящик стола, сделаем общедоступным с помощью Паутины. Принцип при этом остается тем же самым — у вас есть некая дата, а под этой датой располагается список идей и событий, относящихся к этому дню. Один из популярных таких списков вы можете найти по адресу www.bloghop.com.

Создание web-регистратора

Для того чтобы продемонстрировать мощь и функциональные возможности ASP.NET, разработаем приложение web-регистратора с нуля — начиная с единственной чистой web-страницы и последовательно добавляя все новые и новые возможности и давая необходимые пояснения по мере продвижения к цели. Сначала мы продемонстрируем, каким образом можно выводить на экран записи в web-регистратор — по одной или сразу несколько. Затем перейдем к рассмотрению возможностей сортировки записей, их добавления и редактирования, после чего коротко остановимся на вопросах, касающихся безопасности, например, на ограничении доступа к сайту и аутентификации пользователей, зашедших на страницу регистрации. По мере этого продвижения вы познакомитесь со следующими темами:

- Добавление динамического (активного) содержимого (контента) в web-страницы с помощью управляющих элементов web-форм
- Реализация обработчиков событий для каждого управляющего элемента, используя код на C#
- Создание внешне привлекательного пользовательского интерфейса
- Расширение пользовательского интерфейса для поддержки дополнительных функциональных возможностей

Основы создания приложения

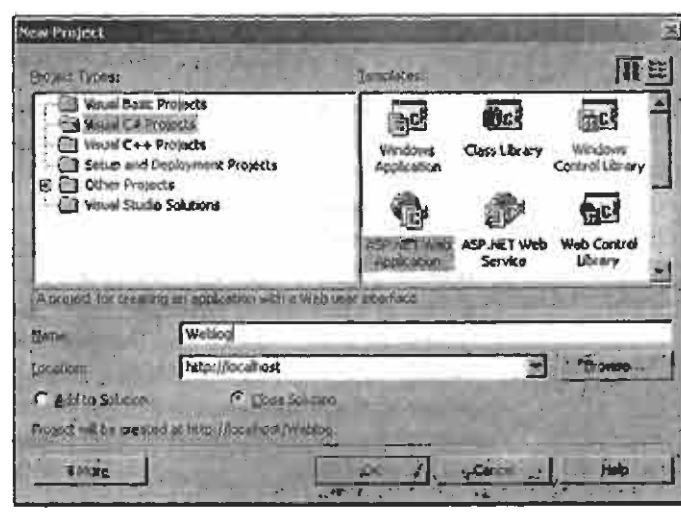
Можно разрабатывать приложения в Visual Studio.NET, используя шаблон ASP.NET Web Application, который создает новый проект и автоматически конфигурирует IIS так, чтобы можно было отлаживать проект по мере его создания.

Первое, что мы собираемся сделать,— это создать на web-сайте простую неизменяемую страницу (другими словами, страницу, в которой нет никаких динамических элементов), с тем чтобы убедиться, что все работает так, как и задумывалось. Эта страница будет создаваться как странница ASP.NET (с расшнреннем `.aspx`), поскольку, несмотря на то, что изначально у этой страницы нет динамических элементов, со временем они появятся.

> *Для ASP-страниц обычно используется расширение* `.asp`*, обозначающее, естественно, Active Server Pages. Для ASP.NET страницы используют расширение* `.aspx`*, что является разновидностью ASP+ — первоначального названия ASP.NET.*

Практикум: создание страницы web-регистратора

1. Откройте Visual Studio.NET и выберите пункт меню File | New | Project. Из списка Project Types выберите Visual C# Projects, а из списка Templates — ASP.NET Web Application. В окно Name введите название — `Weblog` (см. рис. справа).

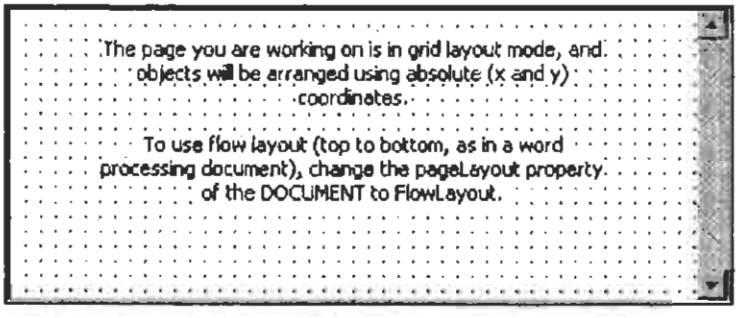


2. Прежде чем нажать OK, обратите внимание на то, что в окне Location находится следующий URL: http://localhost. Это подсказывает, в каком именно месте будет создаваться новый сайт, и, когда мы говорим об Интернете, localhost всегда обозначает ваш собственный компьютер.

3. Для создания нового проекта нажмите OK.

4. Воспользовавшись Solution Explorer, щелкните правой кнопкой мыши на проекте Weblog и выберите пункт меню Add | Add Web Form. Введите имя `Default.aspx` и щелкните мышью на Open.

5. Это должно привести к открытию новой страницы в режиме Design (разработка), однако может и не привести, если вы перед этим экспериментировали с ASP. Если страница находится в режиме Design, то на экране вы должны увидеть следующие два сообщения: "The page you are working on is in grid layout mode, and objects will be arranged using absolute (*x* and *y*) coordinates." (Страница, с которой вы работаете, находится в режиме вывода с сеткой, и объекты будут размещаться на ней с использованием абсолютных координат (*x* и *y*).) и "To use flow layout (top to bottom, as in a word processing document), change the pageLayout property of the DOCUMENT to FlowLayout." (Для использования потокового вывода (сверху вниз, как при обработке документов в текстовых редакторах) измените значение свойства DOCUMENT pageLayout на FlowLayout.), как показано на рисунке слева.



Для того чтобы получить вывод с сеткой, представленный на рисунке, следует выбрать пункт меню View | Design.

6. С помощью программы разработки Designer можно создавать страницы двумя способами — в режиме `GridLayout` или в режиме `FlowLayout`. С точки зрения автора более предпочтительным является режим `FlowLayout`. В окне Properties измените значение свойства `pageLayout` на `FlowLayout`. Точки, которые выводились на странице, исчезнут.

Программа Designer предлагает набор инструментов, которые позволяют облегчить процедуру создания web-сайта. На практике это означает, что создавать web-страницы так же просто, как писать письма с помощью Microsoft Word. Однако предлагаемые в программе Designer инструменты налагают весьма существенные ограничения, поскольку они предназначаются для людей, которые интересуются только разработкой web-интерфейсов, а не программированием. Автор уверен в том, что раз уж вы дочитали книгу до этой главы, то вам нравится создавать программное обеспечение для компьютеров, поэтому будем придерживаться "старой школы" и создавать страницы с помощью HTML. Если вы чувствуете себя неуверенно при программировании на HTML, то, перед тем как двигаться дальше, вы можете просмотреть предыдущую главу и освежить в памяти основы программирования на HTML.

7. Выберите пункт меню **View | HTML Source**. Появится редактор HTML. (В качестве альтернативного варианта в нижней части редактора кода вы сможете обнаружить две маленькие кнопки, помеченные как **Design** и **HTML**. При желании можно пользоваться ими для переключения из одного режима представления в другой.)

8. К скелету HTML-кода добавьте строки, выделенные серым цветом:

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits=Weblog.CDefault" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
   <head>
      <meta name ="GENERATOR" Content="Microsoft Visual Studio 7.0">
      <meta name ="CODE_LANGUAGE" Content="C#">
      <meta name ="vs_defaultClientScript" Content="JavaScript (ECMAScript)">
      <meta name ="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5">
   </head>
   <body>
      <form id="Default" method="post" runat="server">
         <div>
            Disraeli's Weblog
         </div>
         <br>
         <div>
            Copyright &copy; Wrox Press 2001
         </div>
      </form>
   </body>
</html>
```



9. Перед тем как запустить проект, необходимо сообщить VS, что новая страница `Default.aspx` является именно той страницей, которую мы собираемся отлаживать. В Solution Explorer щелкните правой кнопкой мыши на `Default.aspx` и выберите пункт **Set as Start Page**.

Откомпилируйте и запустите проект как обычно. Вы увидите на экране выходной поток, подобный показанному на рисунке слева.

Как это работает

Мы создали новый проект и включили в него одну страницу. VS выполнил большое количество скрытых от наших глаз действий.

Однако в этой странице пока еще нет ничего активного — в ней присутствует только статический текст, который был введен. Познакомимся с тем, каким образом можно включить в эту страницу активные элементы.

Добавление активных элементов

Если вам приходилось когда-нибудь создавать ASP-страницы или разрабатывать динамические web-страницы с использованием каких-либо конкурирующих технологий, то настала пора смело отбросить все, что, как вам кажется, вы знаете

о создании активных страниц. В ASP.NET используется подход, который в корне отличается от всего, с чем вам приходилось сталкиваться ранее.

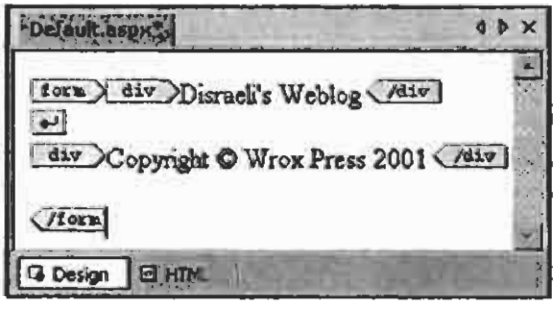
Учитывая тот факт, что Visual Basic оказался таким феноменально успешным, компания Microsoft предлагает использовать для создания web-приложений парадигму, аналогичную использовавшейся при создании приложений Windows Forms. Все строится вокруг управляющих элементов и регулируется событиями.

Представьте себе, например, что есть приложение Windows, которое написано либо на VB, либо на C#. Если возникает желание добавить к этой форме какую-либо кнопку и заставить ее выполнять некоторые функции, то вы используете окно с инструментами, чтобы перенести требуемую кнопку в форму, затем дважды щелкаете на этой кнопке мышью, чтобы создать обработчик событий, и добавляете некий код. Такой подход к программированию является одновременно и основанным на использовании управляющих элементов, и регулирующимся событиями.

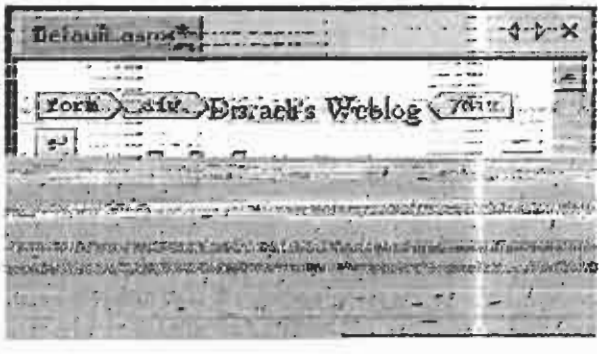
Раньше в Паутине не существовало понятия управляющих элементов. Для всего использовался обычный HTML. Если нужно было нарисовать кнопку, то выполняющемуся на стороне сервера сценарию приходилось отправлять HTML-запрос, чтобы заставить браузер вывести кнопку на экран. Если эта кнопка нажималась, то браузеру приходилось снова посылать запрос странице, на этот раз включая в него дополнительную информацию о нажатии данной кнопки. В этом случае серверный сценарий мог бы среагировать на эту информацию и отправить какую-то новую или обработанную информацию обратно браузеру.

При использовании .NET большая часть этих событий происходит за кулисами. Требуется только нарисовать кнопку в программе разработки Designer, а ASP.NET сам занимается превращением этого управляющего элемента в соответствующий HTML-текст и отправкой его браузеру, после чего браузер рисует его. Если происходит нажатие этой кнопки, то запрос отправляется обратно на сервер, однако мы можем зарегистрировать нажатие как событие, на которое мы в состоянии среагировать. В этом разделе мы увидим, как все это работает на практике.

Практикум: добавление элемента с сообщением о копирайте



1. Если проект активен, завершите его, закрыв окно Internet Explorer.
2. Выберите пункт меню View | Design. Чтобы добавлять на страницу активные элементы, мы должны обязательно находиться в режиме Design View.
3. Для упрощения вывода управляющих элементов выберите в меню пункт View | Details. Вы увидите что-то, подобное показаниому на рисунке слева.



4. Выберите и удалите сообщение о копирайте.
5. Воспользовавшись окном с инструментами, перенесите мышью управляющий элемент `Label` и поместите его между двумя тегами `div`, которые до этого обрамляли сообщение о копирайте. Вы увидите нечто похожее на показаниое на рисунке справа.

Маленькое окошко с символами <?> означает, что созданный тег вообще не является HTML-тегом. Этот тег понятен только для ASP.NET, поэтому он изображается в виде <?>. (Возможно, вы считаете, что компания Microsoft должна была предусмотреть вывод в этом месте маленькой иконки, которая обозначала бы данный управляющий элемент; вполне вероятно, что в следующей версии это будет реализовано именно таким образом!)

6. Важным понять, что концептуально управляющий элемент WebForms Label ничем не отличается от управляющего элемента Windows Forms Label, которым вы уже умеете пользоваться. В окне Properties выбранного вами управляющего элемента вы видите его свойства, в которые можно вносить изменения. Этим мы сейчас и займемся:
   - Присвойте свойству `ID` значение `labelCopyright`
   - Присвойте свойству `Text` значение `(copyright)`
7. Щелкните мышью два раза на странице. Концептуально это то же самое, что и двойной щелчок мышью на форме Windows и приводит к точно таким же результатам! В результате создается обработчик событий, который будет вызываться при загрузке страницы.
8. Введите следующий код в метод `Page_Load()`:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // Какой сейчас год?
    int year = DateTime.Now.Year;
    if(year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;
}
```

Откомпилируйте и запустите проект. (Нельзя щелкнуть правой кнопкой мыши на редакторе кода и выбрать пункт меню View in Browser. Вам придется запустить код так, как вы делали это ранее.) Вы увидите, что никаких изменений не произошло (см. рис. справа).



ОЧЕНЬ ВАЖНО! **Если вы используете программу Outlook или какую-либо другую программу, которая работает с датами, ЗАКРОЙТЕ ЕЕ! Мы собираемся перевести системные часы вперед, и если вы не хотите, чтобы вам немедленно были выведены все напоминания за ближайшие двенадцать месяцев, то лучше закрыть эту программу. Это касается также и других приложений, чувствительных к изменениям даты; если вы не уверены, как они поведут себя при изменении даты, то закройте и их.**



Теперь переведите ваши системные часы на один год вперед. Обновите страницу. Вы увидите картину, подобную представленнй на рисунке слева.

Как это работает

То, что мы сделали, подтверждает факт, что сообщение о копирайте является динамическим. Когда страница создается, она использует метод `DateTime.Now()` для получения текущей даты и вносит соответствующие изменения в сообщение о копирайте.

Но каким образом активный элемент попал на страницу? Чтобы понять это, откройте редактор HTML для данной страницы. Переместившись немного вниз, вы сможете обнаружить следующий тег:

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits=Weblog.CDefault" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
   <head>
      <meta name ="GENERATOR" Content="Microsoft Visual Studio 7.0">
      <meta name ="CODE_LANGUAGE" Content="C#">
      <meta name ="vs_defaultClientScript" Content="JavaScript (ECMAScript)">
      <meta name ="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5">
   </head>
   <body>
      <form id="Default" method="post" runat="server">
         <div>
            Disraeli's Weblog
         </div>
         <br>
         <div>
            <asp:Label  id="labelCopyright"
                        runat="server">(copyright)<asp:Label>
         </div>
      </form>
   </body>
</html>
```

Когда приходит запрос на `.aspx`-страницу, ASP.NET просматривает всю страницу целиком, прежде чем предпринимать какие-либо действия. При этом выявляется, какие составные части страницы являются неизменяемыми, а какие — динамическими. В приведенном выше примере кода выделен тег `<asp:Label>`. ASP.NET знает, что этот тег является активной частью страницы. В результате исключения устанавливается, что оставшаяся часть страницы неизменяемая.

В конечном счете, для ASP.NET необходимо, чтобы вся страница была неизменяемой, поэтому основная роль ASP.NET заключается в превращении динамических элементов страницы в неизменяемый HTML. Большинство современных web-браузеров ограничены в своих возможностях. Основной подход ASP.NET заключается в том, чтобы провести большую часть обработки на сервере и вернуть назад HTML, который может быть выведен браузером. В отдельных случаях ASP.NET допускает отправку браузеру некоторого кода, который должен быть им выполнен. Вероятнее всего, в течение последующих нескольких лет все большая и большая часть работы будет выполняться именно браузером, однако на сегодняшний день

мы предпочитаем, чтобы все выполнялось на сервере, что и происходит в приведенной программе. Посмотрим еще раз на код ASP.NET:

```
<div>
   <asp:Label id="labelCopyright" runat="server">(copyright)<asp:Label>
</div>
```

Такой код фактически получает IE:

```
<div>
   <span id="labelCopyright">Copyright &copy; Disraeli 2001-2002</span>
</div>
```



При создании страниц ASP.NET с помощью Visual Studio.NET, мы по умолчанию используем модель, в которой странница кодируется в двух составляющих. Первая — это `.aspx`-файл, в котором хранится шаблон данной страницы и определяется, в каком именно месте у страницы могут появиться активные элементы. Вторая часть представляет собой файл с расширением `.cs` или `.vb`, в котором содержится код обработчика события (см. рис. слева). Когда производится двукратный щелчок мышью на странице для получения доступа к обработчику события `Load`, мы фактически открываем новый файл.

Исходный файл на C#, известный под названием файла с кодом поддержки, использует то же имя, что и файл с расширением `.aspx`, и неразрывно с ним связан. В нем содержится определение класса, который является производным от класса `System.Web.UI.Page`.

```
/// <summary>
/// Краткое описание класса CDefault.
/// </summary>
public class CDefault : System.Web.UI.Page
```

ASP.NET автоматически добавляет префикс "C" (от английского слова "class") к имени класса, чтобы уменьшить вероятность конфликтов между именами.

Данный класс вызывает событие `Load`, которое исполняется при загрузке страницы. Это происходит после того, как ASP.NET загрузил файл `.aspx` и исследовал его с целью выяснения того, какие динамические элементы имеются на станице.

Представляет интерес и следующий момент. Каждый динамический элемент, определенный на странице, существует также как защищенный член класса. Мы изменили свойство `ID` у управляющего элемента на `labelCopyright`, и именно это является членом, определенным на форме:

```
///summary
/// Краткое описание класса CDefault.
/// </summary>
public class CDefault : System.Web.UI.Page
{
   protected System.Web.UI.WebControls.Label labelCopyright;
```

Это еще одна классическая иллюстрация того, насколько на самом деле Web-Forms и Windows Forms похожи друг на друга. Когда мы рисуем управляющий элемент в форме Windows, то происходит добавление аналогичиого члена. После того как он размещен, мы получаем возможность осуществлять к нему доступ программным путем:

```
private void Page_Load(object sender, System.EventArgs e)
{
   // какой сейчас год?
   int year = DateTime.Now.Year;
   if(year == 2001)
      labelCopyright.Text = "Copyright &copy; Disraeli " + year;
   else
      labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;
}
```

Здесь ASP.NET загрузил .aspx-файл и осуществил поиск динамических элементов. Тем самым ASP.NET имеет в памяти образ страницы в коде HTML с включенными в этот код маркерами в тех местах, где он нашел активный элемент. После этого ASP.NET предоставляет странице и ее управляющим элементам возможность заменить маркеры на их HTML-код. Присваивая значение Text управляющему элементу Label, мы сообщаем этому элементу, что, когда ASP.NET запросит вставку HTML-кода на место маркера в образе страницы, где зарезервировано для него место, управляющий элемент Label выдаст следующий HTML-код:

```
<span id="labelCopyright">Copyright &copy; Disraeli 2001-2002</span>
```

Другими словами, он возьмет значение, которое хранится в свойстве Text, и заключит его в теги .

Поэтапно ASP.NET определит, что все управляющие элементы предоставили свои HTML-коды, составляющие ту часть страницы, за которую они отвечают, и отправит законченный HTML-образ браузеру, который выведет его на экран.

Вхождения web-регистратора

Рассмотрев основной принцип построения страниц ASP.NET, в котором используется парадигма управляющего элемента/события, перейдем к вопросу о том, каким образом этот принцип может быть использован для создания чего-нибудь полезного. Мы создадим отдельную страницу, где будет целиком выводиться вхождение web-регистратора, которое представляет собой список событий и планов на конкретный день.

Одно из преимуществ создания активного web-сайта заключается в том, что не требуется писать код на HTML каждый раз, когда необходимо дополнить или изменить его содержимое. Отсюда следует, во-первых, что добавлять в него новое содержимое теперь могут и люди, ничего не понимающие в HTML, и, во-вторых, что опытные программисты на HTML могут добавлять новое содержание быстро и просто.

Традициоиным способом наполнять сайт новым содержанием является предоставление пользователю некоторой формы, в которую он вводит необходимую информацию. После щелчка мышью на кнопке Save Changes (сохранить изменения) новое содержание появляется на сайте. Этот контент обычно хранится в какой-либо базе данных.

В настоящем примере мы собираемся хранить содержимое web-сайта в отдельных XML-файлах на web-сервере, а не в базе данных. В каждом файле будет храниться одно "вхождение" web-регистратора или описание того, что случилось за определенное время. Мы рассмотрим, каким образом можно создавать и размещать такие вхождения на странице.

Хранение вхождений web-регистратора

Как мы уже говорили, мы собираемся хранить XML-файлы, в которых содержатся вхождения web-регистратора, на сервере. Однако нам придется преодолеть несколько серьезных препятствий, прежде чем мы сумеем достигнуть этого.

Неизвестно, где именно хранятся все файлы только что созданного проекта. Это происходит потому, что для организации хранения файлов VS взаимодействует с web-сервером через нечто, называемое "FrontPage Extensions".

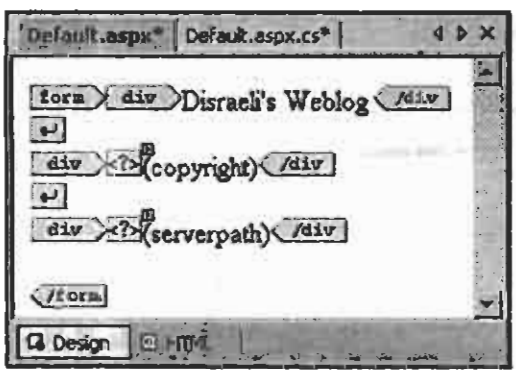
> *FrontPage Extensions имеют весьма пеструю историю. Принцип, положенный в их основу, заключается в том, что они делают внесение изменений в удаленный web-сайт где-то в Интернете таким же простым, как копирование из одной папки в другую, выполняемое на одном и том же компьютере. Исторически сложилось так, что FrontPage Extensions с трудом поддавались конфигурированию администраторами, приводили к возникновению проблем с обеспечением безопасности web-сайта и вообще обладали некоторыми странностями. Можно только надеяться на то, что версия, включенная в .NET, свободна от этих недостатков.*

Это приводит к возникновению первой проблемы, с которой мы сталкиваемся, поскольку в идеале необходимо знать, где именно располагаются эти файлы, с тем чтобы иметь возможность осуществлять их дублирование. К счастью, ASP.NET в состоянии определить их местонахождение.

Практикум: определение местонахождения web-сайта

1. Откройте редактор HTML для файла `Default.aspx` и добавьте в него код, выделенный серым цветом. (Для краткости часть уже существующего кода опущена.)

```
<form id="Default" method="post" runat="server">
   <div>
      Disraeli's Weblog
   </div>
   <br>
   <div>
      <asp:Label  id="labelCopyright"
                  runat="server">(copyright)<asp:Label>
   </div>
   <br>
   <div>
   </div>
</form>
```



2. Выберите пункт меню **View | Design** для открытия программы Designer

3. Воспользовавшись окном с инструментами, поместите с помощью мыши управляющий элемент `Label` между двух новых тегов `div`, как показано на рисунке слева. Присвойте свойству `ID` значение `labelServerPath`, а свойству `Text` — значение `(serverPath)`.

4. Щелкните мышью два раза на основании страницы и снова откройте обработчик событий Load. Добавьте следующий код:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // какой сейчас год?
    int year = DateTime.Now.Year;
    if (year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

    // определение пути к серверу...
    labelServerPath.Text = Server.MapPath("");
}
```



Запустите проект. В результате вы должны увидеть то, что показано на рисунке слева.

Как это работает

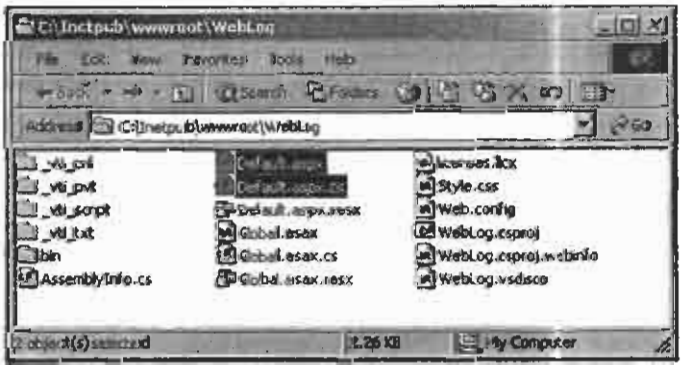
По сравнению с самым первым примером мы добавили новый управляющий элемент, который мы собираемся использовать для выдачи сообщения о пути, по которому располагается наш web-сайт.

Класс `System.Web.UI.Page`, производным от которого является класс `CDefault`, обладает целым рядом свойств, которые помогают нам лучше понимать, что именно требуется от страницы, и позволяют получать доступ к некоторой информации, касающейся среды. Свойство `Server` обладает методом `MapPath`, который позволяет преобразовать виртуальный путь к web-сайту в физический путь к web-серверу. В настоящем примере мы предложили ему преобразовать пустую строку, что означает, что необходимо получить корневую папку web-сайта.

```
private void Page_Load(object sender, System.EventArgs e)
{
    // какой сейчас год?
    int year = DateTime.Now.Year;
    if (year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

    // определение пути к серверу...
    labelServerPath.Text = Server.MapPath("");
}
```

Если мы воспользуемся Internet Explorer и пройдем по пути, выведенному на web-странице, то мы действительно сможем увидеть файлы. На рисунке справа выделены файл с расширением .aspx и файл с расширением .cs, в котором хранится "код поддержки".



Теперь перейдем к созданию XML-файла, который затем может быть использован для хранения вхождения нашего web-регистратора.

Практикум: создание XML-файла, содержащего вхождение

1. Создайте новую папку **Entries** в папке, где находится web-сайт. Например, на компьютере автора эта папка получила следующее название:

   ```
   c:\InetPub\wwwroot\Weblog\Entries
   ```

   До конца главы эта папка будет называться "папкой с вхождениями".

2. Откройте Notepad. Создайте следующий файл:

```
<?xml version="1.0" ?>
<Entry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Title>Hello!</Title>
  <Details>These are the details of the Weblog entry</Details>
</Entry>
```

   XML является чувствительным к регистру, поэтому убедитесь в том, что вы ввели файл в точном соответствии с тем, как он изображен здесь; например, убедитесь, что там, где требуется ввести `<Title>`, вы ввели именно `<Title>`, а не `<title>` или `<TItLe>`.

3. Сохраните файл под именем `Entry.xml` в только что созданной папке с вхождениями.

Вывод вхождений web-регистратора

Теперь есть XML-файл, в котором хранится одно вхождение, и можно пойти еще дальше и создать еще несколько файлов — по одному на каждое вхождение, которое мы хотим выводить в нашем web-регистраторе. Можно также создать класс, который будет загружать эти файлы и представлять хранящиеся в них данные в виде свойств.

Посмотрим теперь, каким образом мы можем загружать эти файлы.

Практикум: загрузка файлов web-регистратора

1. Для загрузки XML-файлов мы планируем включить статический метод в класс, который уже создан Visual Studio.NET и имеет название `Global`. Этот класс представляет собой наиболее подходящее место для размещения общих методов и свойств, которые должны быть доступны всем страницам web-приложения, и который вы сможете обнаружить, заглянув на страницу `Global.asax`.

2. Чтобы найти класс `Global` с помощью Solution Explorer, щелкните правой кнопкой мыши на файле `Global.asax` (см. рис. справа) и выберите пункт меню **View Code**.

3. Вы можете заметить, что класс `Global` является производным от `System.Web.HttpApplication`. В этом классе описаны некоторые события, на которые мы можем реагировать, когда наступают определенные события уровня приложения. (В ASP.NET "уровень приложения" может считаться "уровнем web-сайта".)



4. Добавьте следующий член в класс Global:

```
/// <summary>
/// Краткое описание класса Global.
/// </summary>
public class Global : System.Web.HttpApplication
{
    // члены...
    public static String EntryFilePath;
```

Обратите внимание на то, что мы используем класс String (заглавная S), а не ключевое слово C# string (строчная s). Это сделано именно таким образом, поскольку необходимо, чтобы этот класс видимым образом подчинялся требованиям Common Language Runtime, а если воспользоваться ключевым словом C# string, то оно внутренне будет преобразовываться в System.String — мы просто хотим сэкономить на этой излишней обработке.

5. Добавьте следующий код в метод Application_Start():

```
protected void Application_Start(Object sender, EventArgs e)
{
    // присваивание значения общему члену пути к папке вхождений...
    EntryFilePath = Server.MapPath("Entries");
}
```

6. С помощью Solution Explorer создайте новый класс с именем Entry. Прежде всего добавьте следующие операторы using в самое начало файла:

```
using System;
using System.IO;
using System.Xml.Serialization;
```

7. Затем добавьте следующие члены и соответствующие им свойства:

```
/// <summary>
/// Краткое описание класса Entry.
/// </summary>
public class Entry
{
    // члены...

    private DateTime _timestamp;
    private String _title;
    private String _details;

    [XmlIgnore()] public DateTime Timestamp
    {
        get
        {
            return _timestamp;
        }
        set
        {
            _timestamp = value;
        }
    }

    public String Title
    {
        get
        {
            return _title;
        }
```

```
        set
        {
            _title = value;
        }
    }

    public String Details
    {
        get
        {
            return _details;
        }
        set
        {
            _details = value;
        }
    }
```

Названия этих свойств совпадают с именами вхождений XML-файла, который был нами создан. Это сделано совершенно умышленно!

8. Переключитесь обратно на редактор кода для файла `Global.asax.cs`. Добавьте ссылки на те же самые пространства имен в верхнюю часть файла:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Web;
using System.Web.SessionState;
using System.IO;
using System.Xml.Serialization;
```

9. Теперь добавьте в класс `Global` следующий статический метод:

```
// LoadEntry - загрузка вхождения с диска...
public static Entry LoadEntry(String filename)
{
    // у нас имеется название, но нам необходим путь...
    String filepath = EntryFilePath + "\\" + filename;

    // открытие файла...
    FileStream file = new FileStream(filepath, FileMode.Open);

    // создание сериализатора...
    XmlSerializer serializer = new XmlSerializer(typeof(Entry));
    Entry newEntry = (Entry)serializer.Deserialize(file);

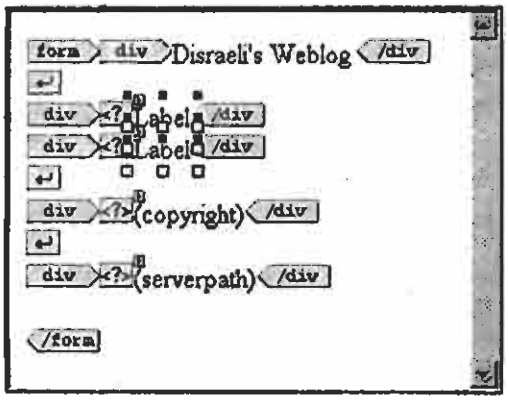
    // закрытие файла...
    file.Close();

    // возврат вхождения...
    return newEntry;
}
```

10. У нас появилась возможность создавать новые объекты `Entry` и присваивать их свойствам `Title` и `Details` данные, хранящиеся в XML-файле. Теперь необходимо создать некоторые управляющие элементы на странице `Default.aspx`, с тем чтобы увидеть результаты этого процесса.

11. Откройте редактор HTML для Default.aspx. Добавьте следующий код:

```
<form id="Default" method="post" runat="server">
   <div>
      Disraeli's Weblog
   </div>
   <br>
   <div>
   </div>
   <div>
   </div>
   <br>
   <div>
      <asp:Label id="labelCopyright" runat="server">(copyright)<asp:Label>
   </div>
...
</form>
```



12. Переключитесь в Designer для Default.aspx. В каждый из двух появившихся элементов div добавьте по новому управляющему элементу Label, как показано на рисунке слева.

- Для первого управляющего элемента присвойте свойство ID значению lalbelEntryTitle. Присвойте свойство Text значению (entryTitle).
- Для второго управляющего элемента присвойте свойство ID значению lalbelEntryDetails. Присвойте свойство Text значению (entryDetails).

13. Щелкните два раза мышью на странице, чтобы открыть обработчик событий Load. Измените последнюю строку, чтобы в ней использовалось общее свойство Global.EntryFilePath:

```
private void Page_Load(object sender, System.EventArgs e)
{
   // какой сейчас год?
   int year = DateTime.Now.Year;
   if(year == 2001)
      labelCopyright.Text = "Copyright &copy; Disraeli " + year;
   else
      labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

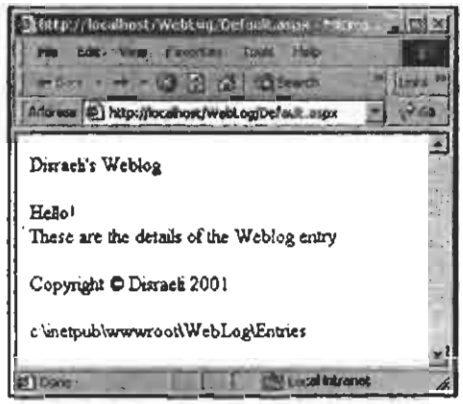
   // задание пути к серверу...
   labelServerPath.Text = Global.EntryFilePath;
}
```

14. Добавьте этот код, чтобы использовать общий метод в Global для создания нового объекта Entry на основе данных, хранящихся в файле Entry.xml, а затем наполните те управляющих элемента, которые были только что добавлены:

```
private void Page_Load(object sender, System.EventArgs e)
{

   ...
   // определение пути к серверу...
   labelServerPath.Text = Global.EntryFilePath;
```

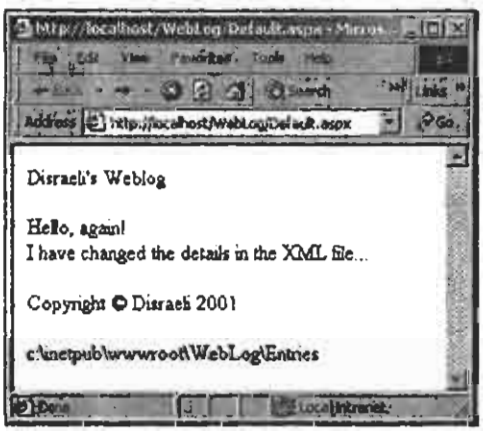
```
    // загрузка вхождения с диска...
    Entry entry = Global.LoadEntry("Entry.xml");
    labelEntryTitle.Text = entryTitle;
    labelEntryDetails.Text = entryDetails;
}
```



Запустите проект. Вы увидите то, что изображено на рисунке справа.

15. Чтобы убедиться, что изменения загружаются с диска, воспользуйтесь Notepad, чтобы еще раз отредактировать файл `Entry.xml`. Измените данные так, чтобы они приняли следующий вид:

```
<?xml version="1.0"?>
<Entry xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Title>Hello, again!</Title>
    <Details>I've changed the details in the XML file...</Details>
</Entry>
```



16. Сохраните этот файл и обновите страницу в Internet Explorer. Вы увидите внесенные изменения (см. рис. слева).

Как это работает

Первое, что мы сделали в `Global.asax`,— это повторно воспользовались методом `Server.MapPath` для получения физического пути к виртуальной директории `Entries`, который можно использовать в программе. После получения первого запроса на `.aspx`-страницу вызывается метод `Application_Start()`, позволяющий задавать глобальные данные приложения (или глобальные данные сайта). В данном случае мы сохраняем физический путь в общем члене `EntryFilePath`:

```
protected void Application_Start(Object sender, EventArgs e)
{
    // присвоение значения физического пути общему члену...
    EntryFilePath = Server.MapPath("Entries");
}
```

Теперь мы подходим к статическому методу `LoadEntry()`, который загружает XML-файлы с данными из папки с вхождениями и переносит содержимое в определенный нами класс `Entry`. Для этого используем класс `XmlSerializer` из пространства имен `System.Xml.Serialization`. Далее вызывается его метод `Deserialize()`, который принимает XML-строку из файла и преобразовывает (десериализует) находящиеся в ней данные, чтобы использовать их для присвоения свойствам объекта, которые хранятся в экземпляре класса `Entry`.

```
// LoadEntry - загрузка вхождения с диска...
public static Entry LoadEntry(String filename)
{
    // у нас есть название, но нам необходим путь...
    String filepath = EntryFilePath + "\\" + filename;

    // открытие файла...
    FileStream file = new FileStream(filepath, FileMode.Open);
```

```
    // создание сериализатора и использование его для наполнения свойств
    // вновь созданного объекта Entry...
    XmlSerializer serializer = new XmlSerializer(typeof(Entry));
    Entry newEntry = (Entry)serializer.Deserialize(file);

    // закрытие файла...
    file.Close();

    // возврат вхождения...
    return newEntry;
}
```

Из Default.aspx.cs мы опять настраиваем обработчик событий Load. На этот раз мы исходим из предположения, что XML-файл всегда называется Entry.xml; после этого мы присваиваем свойство Text каждому из управляющих элементов следующим образом:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // какой сейчас год?
    int year = DateTime.Now.Year;
    if(year == 2001)
        labelCopyright.Text = "Copyright &copy; Disraeli " + year;
    else
        labelCopyright.Text = "Copyright &copy; Disraeli 2001-" + year;

    // определение пути к серверу...
    labelServerPath.Text = Global.EntryFilePath;

    // загрузка вхождения с диска...
    Entry entry = Global.LoadEntry("Entry.xml");
    labelEntryTitle.Text = entryTitle;
    labelEntryDetails.Text = entryDetails;
}
```

Вывод времени

Теперь нам предстоит вывести время рядом с вхождением. Для этой цели мы собираемся использовать значение времени, когда этот файл был сохранен последний раз, и поместить это значение в свойство вхождения Timestamp. После этого мы сможем немного изменить код, чтобы он сначала выводил дату и время, а затем всю остальную информацию вхождения.

Практикум: вывод времени

1. Для определения времени мы воспользуемся значением, когда данное вхождение изменялось последний раз. В большинстве случаев это будет время последней записи вхождения на диск.

2. Воспользуйтесь Solution Explorer, щелкните правой кнопкой мыши на Global.asax и выберите пункт меню View Code. Найдите метод LoadEntry и добавьте следующий код:

```
// LoadEntry – загрузка вхождения с диска...
public static Entry LoadEntry(String filename)
{
    . . .

    //закрытие файла
    file.Close ();

    // обновление даты и времени...
    newEntry.Timestamp = new FileInfo(filepath).LastWriteTime;
```

```
        // возврат вхождения...
        return newEntry;
    }
```

3. Открываем редактор для `Default.aspx` и вносим следующие изменения:

```
private void Page_Load(object sender, System.EventArgs e)
{
    ...
    // определение пути к серверу...
    labelServerPath.Text = Global.EntryFilePath;
    // загрузка вхождения с диска...
    Entry entry = Global.LoadEntry("Entry.xml");
    labelEntryTitle.Text = entryTitle;
    labelEntryDetails.Text = entry.Timestamp.ToString("ddd") + ", " +
        entry.Timestamp.ToLongDateString() + " - " + entryDetails;
}
```



Запустите проект. Теперь вы должны увидеть на экране время, как показано на рисунке слева.

Как это работает

У объекта `Entry` уже есть соответствующий член для хранения даты и временн. Здесь используется атрибут `XmlIgnore`, для того чтобы `XmlSerializer` осуществлял соответствующие действия, именно поэтому нам не приходится включать их в XML-файл. Все, что помечено этим атрибутом, не будет ни сохраняться в XML-файле, ни загружаться из XML-файла.

```
/// <summary>
/// Краткое описание класса Entry.
/// </summary>
public class Entry
{
    // члены...
    [XmlIgnore()] public DateTime Timestamp;
    public String Title;
    public String Details;
```

При создании объекта `Entry` мы воспользовались объектом `System.IO.FileInfo`, в котором хранится дата последнего сохранения файла:

```
// LoadEntry - загрузка вхождения с диска...
public static Entry LoadEntry(String filename)
{
    ...
    // закрытие файла...
    file.Close();
    // обновляем дату и время...
    newEntry.Timestamp = new FileInfo(filepath).LastWriteTime;
    // возврат вхождения...
    return newEntry;
}
```

Когда мы подошли к фактическому выводу страницы, мы воспользовались двумя методами для преобразования даты в строки. Первый вызов `ToString()` оформлен так, чтобы выводить название дня недели. Второй вызов `ToLongDateString()`

обрабатывает остальную часть длинной даты, используя локальные установки компьютера. В США сначала будет выводиться месяц, а затем число, в Великобритании — сначала число, а потом месяц и т. д. Это хорошая идея — использовать локализованную версию функции для работы с датами, в противном случае ваши пользователи могут быть сбиты с толку.

```
private void Page_Load(object sender, System.EventArgs e)
{
   ...

   // определение пути к серверу...
   labelServerPath.Text = Global.EntryFilePath;

   // загрузка вхождения с диска...
   Entry entry = Global.LoadEntry("Entry.xml");
   labelEntryTitle.Text = entryTitle;
   labelEntryDetails.Text = entry.Timestamp.ToString("ddd") + ", " +
      entry.Timestamp.ToLongDateString() + " - " + entryDetails;
}
```

Улучшение внешнего вида

Пока страница имеет весьма уродливый внешний вид. HTML (с точки зрения сегодняшнего дня) — это устаревшая технология, которая прежде всего предназначалась для организации доступа к однообразным научным документам. Придание документам обновленного и современного вида возлагается, в основном, на пользователей Паутины, поэтому если мы хотим, чтобы документы выглядели соответствующим образом, то придется позаботиться об этом самим.

Существует целый ряд технологий, которые можно использовать для улучшения представления страницы, но в данном случае проще всего воспользоваться технологией **Cascading Style Sheets (CSS — каскадные таблицы стилей)**. CSS — это огромная тема, поэтому мы не будем останавливаться на ней очень подробно, будет достаточно сказать, что CSS — это специальным образом отформатированный текстовый файл, в котором хранятся все шрифты, цвета и стили, необходимые браузеру для представления HTML в приятной для глаза форме.

Практикум: улучшение внешнего вида

1. Первое, что требуется сделать,— это создать таблицу стилей, которая будет распространяться на все страницы нашего сайта. Для этого войдите в Solution Explorer, щелкните правой кнопкой мыши на проекте Weblog и выберите пункт меню **Add | Add New Item**. Найдите в списке **Templates** вхождение **Style Sheet**. Выберите его и присвойте имя `style.css`. Щелкните мышью на **Open**.

2. Добавьте следующий код в новую таблицу стилей:

```
body
{
   padding-right: 0px;
   padding-left: 0px;
   font-size: 8pt;
   padding-bottom: 0px;
   margin: 0px;
   padding-top: 0px;
   font-family: Verdana, Arial;
}
```

```
.header
{
   padding-right: 5px;
   padding-left: 5px;
   padding-bottom: 10px;
   padding-top: 10px;
   background-color: #000099;
   font-weight: bold;
   font-size: 14pt;
   color: white;
}
.normal
{
   padding-right: 5px;
   padding-left: 5px;
   font-size:8pt;
}
.normalHeading
{
   padding-right: 5px;
   padding-left: 5px;
   font-size:12pt;
   font-weight: bold;
}
.entryTitle
{
   padding-right: 5px;
   padding-left: 5px;
   padding-bottom: 1px;
   padding-top: 1px;
   font-weight: bold;
   font-size: 10pt;
   color: white;
   background-color: #66cc99;
}
.entryDate
{
   font-weight: bold;
   color: #333399;
   font-size: 8pt;
}
.entry
{
   padding-right: 5px;
   padding-left: 5px;
   padding-top: 2px;
   font-size: 8pt;
}
```

Если вы не хотите вводить все это вручную, возьмите файл `style.css` из загружаемого кода; если он до сих пор не включен в проект, щелкните правой кнопкой мыши на узле **Weblog project** в Solution Explorer, выберите пункт меню **Add | Add Existing Item** и добавьте файл с расширением `.css`.

3. Чтобы воспользоваться этой таблицей стилей, необходимо привязать ее к странице. Откройте редактор HTML для файла `Default.aspx` и добавьте следующую строку в верхнюю часть файла:

```
<html>
   <head>
      <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
      <meta content="C#" name="CODE_LANGUAGE">
```

```
        <meta content="JavaScript (ECMAScript)" name="vs_defaultClientScript">
        <meta content="http://schemas.microsoft.com/intellisense/ie5"
              name="vs_targetSchema">
        <link rel="stylesheet" href="style.css">
    </head>
```

4. Внесите изменения в выделенные строки и добавьте новый код, где это необходимо:

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits="WebLog.CDefault" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
    <head>
        <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
        <meta content="C#" name="CODE_LANGUAGE">
        <meta content="JavaScript (ECMAScript)" name="vs_defaultClientScript">
        <meta content="http://schemas.microsoft.com/intellisense/ie5"
              name="vs_targetSchema">
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <form id="Default" method="post" runat="server">
            <div class="header">
                Disraeli's Weblog
            </div>
            <br>
            <div class="entryTitle">
                <asp:label id="labelEntryTitle" runat="server"
                     (entryTitle)</asp:label>
            </div>
            <div class="entry">
                <asp:Label id="labelEntryDetails" runat="server"
                    (entryDetails)</asp:label>
            </div>
            <br>
            <hr color="#000000">
            <div class="normal">
                <asp:label id="labelCopyright" runat="server">
                    (copyright)</asp:label>
            </div>
            <br>
            <div class="normal">
                <asp:Label id="labelServerPath" runat="server">
                    (serverpath)</asp:Label>
            </div>
        </form>
    </body>
</html>
```

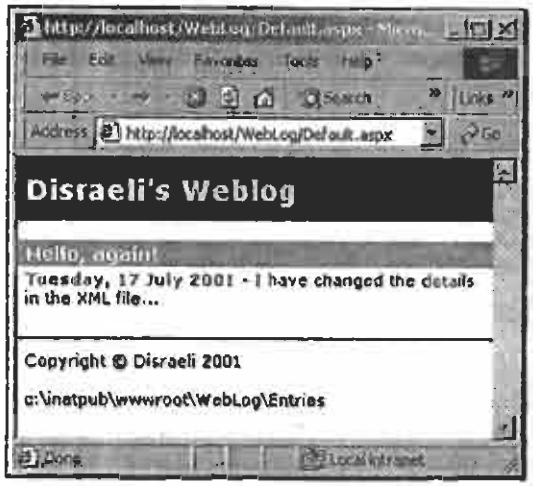
5. Последнее изменение в коде, которое необходимо внести, касается `Default.aspx.cs`. Откройте этот файл в редакторе и внесите в метод `Page_Load()` следующие изменения:

```
private void Page_Load(object sender, System.EventArgs e)
{
    . . .

    // определение пути к серверу...
    labelServerPath.Text = Global.EntryFilePath;

    // загрузка вхождения с диска...
    Entry entry = Global.LoadEntry("Entry.xml");
    labelEntryTitle.Text = entry.Title;
```

```
            labelEntryDetails.Text = "<font class=entryDate>" +
                entry.Timestamp.ToString("ddd") + ", " +
                entry.Timestamp.ToLongDateString() + " <font> " +
                entryDetails;
        }
```



После компиляции и запуска проекта вы должны увидеть то, что изображено на рисунке слева.

Как это работает

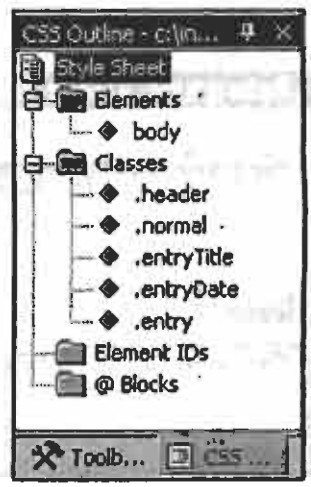
Каскадные таблицы стилей позволяют определять стили, которые впоследствии будут применяться ко всем элементам, расположенным на странице. Эти стили создаются в виде файлов с расширением `.css`, подобных созданному нами файлу `style.css`.

Обсуждение технологии CSS выходит за рамки настоящей книги, но основная ее суть сводится к следующему: есть возможность либо изменить стиль стандартных HTML-элемеитов, таких как `<body>` или `<ul>`, либо описать новые классы, которые в терминологии CSS означают группу стилей, объединенных в некоторую логическую единицу. Чтобы модифицировать уже существующий элемент, требуется просто указать имя этого элемента, а затем добавить соответствующий CSS-код. Именно таким образом мы переопределяем то, каким образом выводится на экран тег `<body>`:

```
body
{
    padding-right: 0px;
    padding-left: 0px;
    font-size: 8pt;
    padding-bottom: 0px;
    margin: 0px;
    padding-top: 0px;
    font-family: Verdana, Arial;
}
```

Новым классам может даваться произвольное имя, однако оно должно предваряться точкой. Вот как создается класс `header`:

```
.header
{
    padding-right: 5px;
    padding-left: 5px;
    padding-bottom: 10px;
    padding-top: 10px;
    background-color: #000099;
    font-weight: bold;
    font-size: 14pt;
    color: white;
}
```



Чисто технически от вас не требуется понимания CSS для того, чтобы вы могли редактировать его в Visual Studio.NET. Можете быть совершенно спокойны — перед вами действительно код таблицы стилей. Если выбранным в настоящий момент документом является таблица стилей, то с помощью пункта меню View | Other Windows | Document Outline вы можете войти в редактор CSS (см. рис. справа).

Если щелкнуть правой кнопкой мыши на каком-либо из элементов или классов и выбрать пункт Build Style, то откроется диалоговое окно, с помощью которого можно внести некоторые незначительные изменения в CSS-код на интуитивном уровне.

После того, как таблица стилей построена, ее необходимо привязать к соответствующей странице. Это позволит браузеру получить информацию о том, что требуется загрузить описанные стили и применить их к документу. Это осуществляется посредством элемента link:

```
<html>
  <head>
    <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
    <meta content="C#" name="CODE_LANGUAGE">
    <meta content="JavaScript (ECMAScript)" name="vs_defaultClientScript">
    <meta content="http://schemas.microsoft.com/intellisense/ie5"
          name="vs_targetSchema">
    <link rel="stylesheet" href="style.css">
  </HEAD>
```

Для того чтобы задать класс, который следует использовать с элементом на странице, применяется атрибут class. Он позволяет передать браузеру информацию о том, какой именно стиль должен применяться к данному элементу. Мы сообщаем информацию о том, что элемент div, содержащий текст Disraeli's Weblog, должен использовать класс header следующим образом:

```
<div class="header">
  Disraeli's Weblog
</div>
```

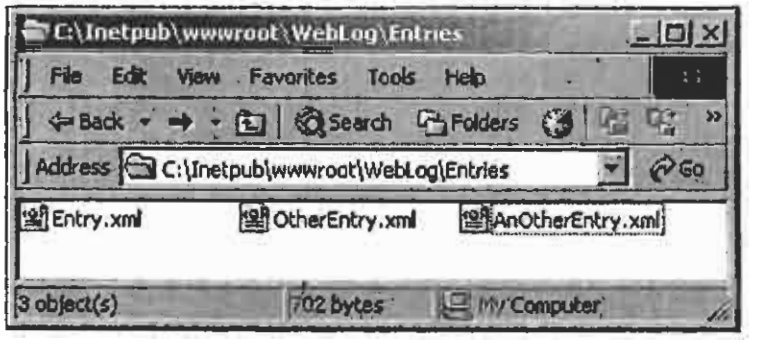
Мы будем использовать данную таблицу стилей на протяжении оставшейся части настоящей главы, хотя мы не будем разбираться в этих стилях более подробно.

Вывод списков вхождений

К настоящему моменту мы научились выводить на экран отдельные вхождения web-регистратора, но как нам быть в том случае, если требуется вывести несколько вхождений на одной о той же странице?

Вывод списков — довольно распространенная задача, часто стоящая перед активными web-сайтами независимо от того, требуется вывести список продаваемой продукции на сайте электроной коммерции или записи пользователя, хранящиеся в базе данных. В ASP.NET предусмотрен целый ряд способов, которые облегчают вывод списков, и некоторые из этих способов мы рассмотрим в настоящем разделе.

Практикум: вывод списков вхождений web-регистратора



1. Для того чтобы иметь возможность просматривать список вхождений web-регистратора, необходимо, чтобы в папке Entries находилось более одного вхождения! Используя Internet Explorer и Notepad, создайте какие-нибудь дополнительные вхождения. Можете назвать их произвольным образом (см. рис. слева).

2. В классе `Global` потребуется определить новый статический метод, который будет возвращать список всех вхождений. Откройте в редакторе кода файл `Global.asax` и добавьте следующий код:

```
// LoadAllEntries - осуществляет загрузку с диска всех вхождений...
public static Entry[] LoadAllEntries()
{
   // определение пути, по которому располагаются вхождения...
   DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

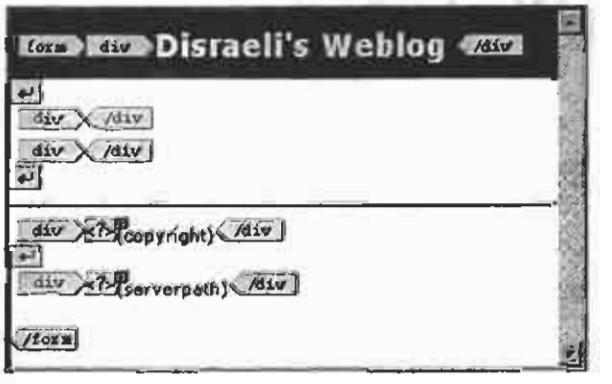
   // получение списка файлов...
   FileInfo[] files = entryFolder.GetFiles();

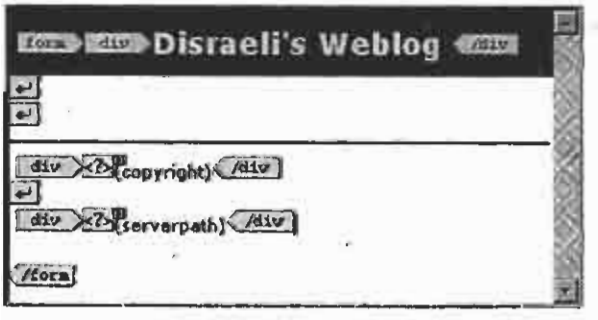
   // создание массива вхождений...
   Entry[] entries = new Entry[files.Length];

   // просмотр в цикле всех файлов и их загрузка...
   int index = 0;
   foreach(FileInfo file in files)
   {
      entries[index] = LoadEntry(file.Name);
      index++;
   }

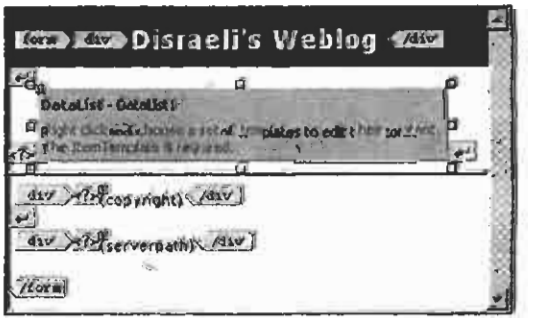
   // возврат списка...
   return entries;
}
```

3. Теперь требуется исключить из файла `Default.aspx` код, ответственный за вывод двух управляющих элементов. Откройте программу-разработчик и удалите два управляющих элемента `Label`. Это должно выглядеть, как показано на рисунке справа.





4. Удалите два тега `div`, как показано на рисунке слева.



5. Воспользовавшись окном с инструментами, перенесите управляющий элемент `DataList` и разместите его между двумя символами "возврат каретки". Результат представлен на рисунке справа.

6. Управляющий элемент `DataList` довольно мощный, но он требует выполнения огромного количества различных действий и манипуляций над UI. В нем используется полная

привязка данных, т. е. в методе `Page_Load()` приходится передавать этому управляющему элементу массив объектов `Entry`. Он проходит весь список в цикле и на каждой итерации представляет набор описанных нами управляющих элементов. Каждый из этих управляющих элементов привязан посредством данных к некоторому свойству класса `Entry`. Поэтому можно добавить в управляющий элемент `DataList` управляющий элемент `Label` и привязать его свойство `Text` к свойству `Title` и т. д.

7. Прежде чем предпринимать какие бы то ни было шаги, следует изменить значение свойства `ID` управляющего элемента `DataList` на `dataListEntries`.

8. Щелкните правой кнопкой мыши на элементе `DataList` и выберите пункт меню **Edit Template | Item Template**. Это переведет управляющий элемент в особый режим, в котором мы получаем возможность управлять тем, что конкретно будет происходить на каждой итерации по массиву объектов `Entry`.



9. Воспользовавшись окном с инструментами, перенесите управляющий элемент `Label` и разместите его в пустом пространстве, расположенном непосредственно под строкой `ItemTemPlate` (см. рис. слева).

   Любые управляющие элементы, которые располагаются внутри вхождения `ItemTemPlate`, будут выводится на экран при каждой очередной итерации по массиву. Поэтому, если в массиве есть, например, три объекта `Entry`, в конечном итоге окажется три управляющих элемента `Label`.

10. При выбранном элементе `Label` с помощью окна **Properties** измените значение свойства `CssClass` на `entryTitle`. Внешний вид управляющего элемента должен измениться в соответствии выбранным стилем.

11. Найдите вхождение `DataBindings` и выберите его. Щелкните мышью на многоточии (...), расположенном рядом с ним, для перехода в окно для привязки данных.

    В списке, расположенном с левой стороны, перечислены все свойства управляющего элемента `Label`, которые допускают привязку к данным. Из него следует выбрать свойство `Text`. После этого все действия, которые мы будем производить на правой стороне окна, будут относиться к настройкам привязки к данным именно свойства `Text`.

12. Выберите переключатель **Custom binding expression** и введите следующий код в текстовое окно, находящееся под этим переключателем:

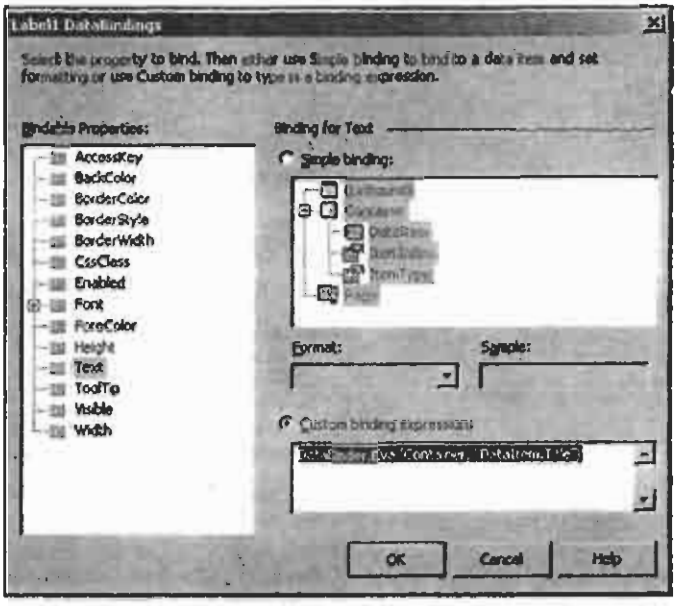
```
DataBinder.Eval(Container, "DataItem.Title")
```

Результат проиллюстрирован на рисунке (см. стр. 705, справа).

13. Для сохранения привязки к данным щелкните мышью на OK.

14. Необходимо проверить, работает ли то, что мы уже сделали. В случае положительного результата проверки можно перейти к выводу даты и другой информации.





Щелкните два раза мышью на странице, чтобы открыть обработчик событий Load. Для того, чтобы получить картину, изображенную на рисунке слева, нужно откомпилировать и запустить проект после добавления в обработчик событий Load следующего кода:

```
private void Page_Load(object sender, System.EventArgs e)
{
   ...

   // определение пути к серверу...
   labelServerPath.Text = Global.EntryFilePath;

   // загрузка всех вхождений, имеющихся на диске...
   Entry[] entries = Global.LoadAllEntries();
   datalistEntries.DataSource = entries;
   datalistEntries.DataBind();
}
```

Как это работает

Код, создающий массив на диске из всех объектов Entry, крайне прост. Для этого требуется лишь обратиться к объекту System.IO.DirectoryInfo для формирования списка всех объектов System.IO.DirectoryInfo для папки Entries и пройти по списку в цикле, используя метод LoadEntry.

```
// LoadAllEntries — осуществляет загрузку с диска всех вхождений...
public static Entry[] LoadAllEntries()
{
   // определение пути, по которому располагаются вхождения...
   DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

   // получение списка файлов...
   FileInfo[] files = entryFolder.GetFiles();

   // создание массива вхождений...
   Entry[] entries = new Entry[files.Length];

   // просмотр в цикле всех файлов и их загрузка...
   int index = 0;
   foreach(FileInfo file in files)
```

```
        {
            entries[index] = LoadEntry(file.Name);
            index++;
        }

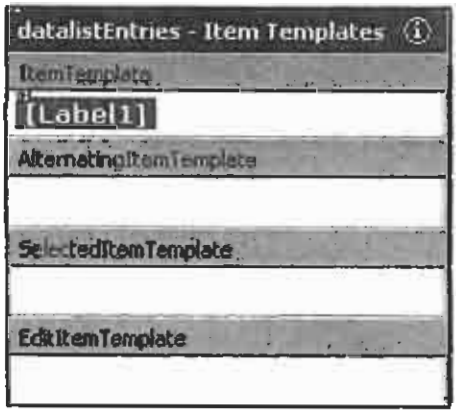
        // возврат списка...
        return entries;
    }
```

После того как мы получаем список в `Page_Load()`, мы передаем его управляющему элементу `DataList` и вызываем метод `DataBind()`:

```
private void Page_Load(object sender, System.EventArgs e)
{
    ...

    // определение пути к серверу...
    labelServerPath.Text = Global.EntryFilePath;

    // загрузка всех вхождений, имеющихся на диске...
    Entry[] entries = Global.LoadAllEntries();
    datalistEntries.DataSource = entries;
    datalistEntries.DataBind();
}
```



Это равносильно просьбе, обращенной к управляющему элементу `DataList`, чтобы пройти по очереди по всем объектам `Entry`, следуя правилам, определенным в шаблоне. Единственным правилом, описанным нами в шаблоне, является правило, согласно которому один раз на каждой итерации требуется вывести управляющий элемент `Label` — при этом свойству `Text` элемента `Label` должно быть присвоено текущее значение свойства `Title` того объекта `Entry`, рассматриваемому в текущий момент (см. рис. слева).

Вывод подробной информации

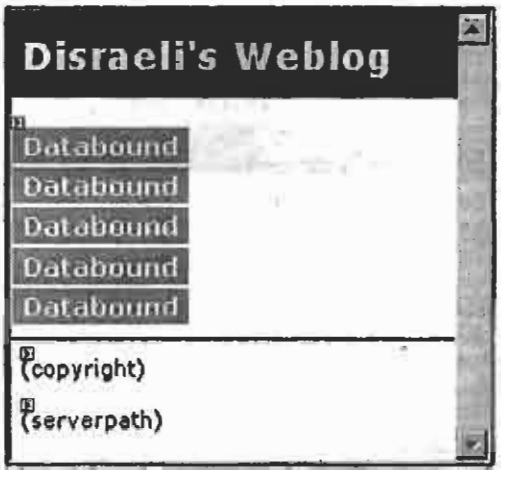
Теперь можно выводить заголовок, но как быть с остальной информацией? В данном разделе мы рассмотрим, каким образом можно выводить подробную информацию и дату.

Практикум: вывод остальной части вхождения web-регистратора

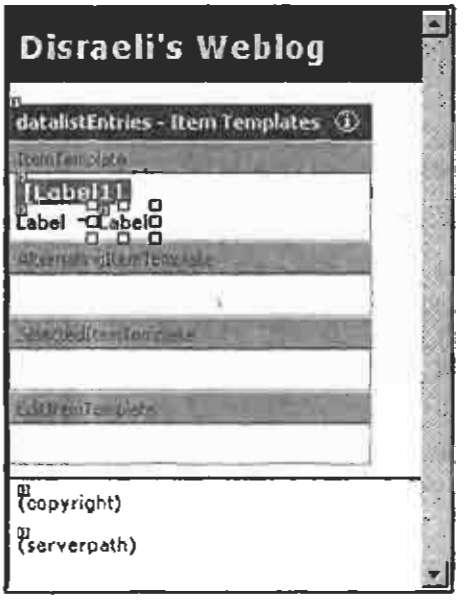
1. Способ, с помощью которого мы выводили дату вхождения web-регистратора, был очень сложным. Поскольку управляющий элемент `DataList` предназначен специально для работы с общими свойствами, мы создадим такое свойство объекта `Entry`, которое будет возвращать дату в виде строки. Откройте редактор кода для `Entry` и введите следующий код:

```
public String TimestampAsString
{
    get
    {
        return Timestamp.ToString("dddd") + ", " +
                                Timestamp.ToLongDateString();
    }
}
```

2. Откройте **Designer** для `Default.aspx`. Управляющий элемент `DataList` может перейти в такое состояние, в котором его нельзя редактировать. Это выглядит так, как показано на рисунке справа.



3. Щелкните правой кнопкой мыши на управляющем элементе и снова выберите пункт меню **Edit Template | Item Template**. Установите курсор после зеленого управляющего элемента `Label`, нажмите клавишу *Shift* и, не отпуская ее, нажмите клавишу *Return*. В результате вместо символа возврата каретки будет добавлен символ новой строки, что приведет к появлению очень большого пустого пространства между заголовком и самим вхождением.



4. Добавьте еще один управляющий элемент `Label` под уже существующим элементом `Label` зеленого цвета. После этого добавьте пробел, затем тире, еще один пробел, а затем еще один — последний — элемент `Label`, как показано на рисунке слева.

   Маленькая зеленая стрелка в верхнем левом углу каждого элемента обозначает управляющий элемент.

5. Выберите первый новый управляющий элемент `Label`. Присвойте его свойству `CssStyle` значение `entryDate`.

6. Выберите свойство `DataBindings`. Щелкните мышью на многоточии и откройте окно **DataBindings**. Убедитесь, что в списке, расположенном слева, выбрано свойство `Text`, и щелкните мышью на переключателе **Custom binding expression**. Добавьте следующий код:

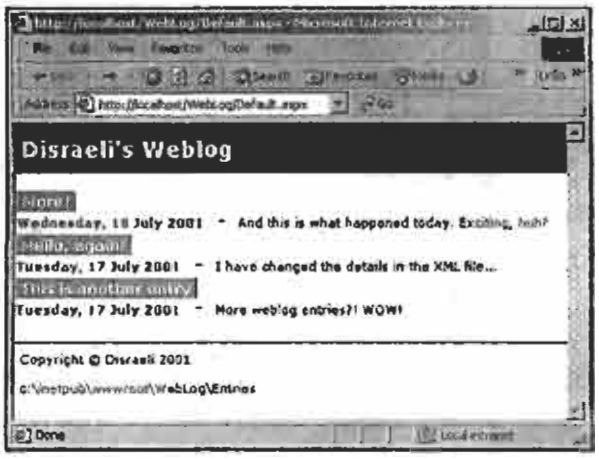
```
DataBinder.Eval(Container, "DataItem.TimestampAsString")
```

7. Щелкните мышью на **OK**, для того чтобы сохранить привязку.

8. Выберите второй из двух новых управляющих элементов `Label`. Присвойте его свойству `CssStyle` значение `entry`.

9. Найдите и выберите свойство `DataBindings`. Щелкните мышью на многоточии и откройте окно **DataBindings**. Убедитесь, что в списке, расположенном с левой стороны, выбрано свойство `Text`, и щелкните мышью на переключателе **Custom binding expression**. Добавьте следующий код:

```
DataBinder.Eval(Container, "DataItem.Details")
```



Откомпилируйте и запустите проект. Вы увидите приблизительно то, что изображено на рисунке слева.

Как это работает

Поскольку при работе с управляющим элементом `DataList` проще всего использовать свойства, мы создали свойство, доступное в режиме "только чтение", с именем `TimestampAsString`, которое должно возвращать отформатированную версию свойства `Timestamp`. Мы не включили в это свойство предложения `set`, поскольку это предполагало бы, что тот, кто использует данный объект, может попросить нас преобразовать это значение из строки в `System.DateTime`. И хотя в этом также нет ничего недостижимого, такая задача выходит за рамки работы, которую мы собираемся возложить на данный объект.

```
public String TimestampAsString
{
   get
   {
      return Timestamp.ToString("dddd") + ", " +
                              Timestamp.ToLongDateString();
   }
}
```

После добавления этого свойства включение дополнительных элементов `Label` в шаблон свелось к простому повторению действий, выполненных ранее.

Важное замечание об управляющем элементе DataList и общих членах

Выражение, введенное в поле **Custom binding expression** элемента `DataList`, будет работать только со свойствами. Оно не сможет работать с общими членами. Если у вас есть такой код:

```
public class Entry
{
   // члены...
   Public String Title;
```

и вы попытаетесь выполнить привязку `Title` с помощью выражения:

```
DataBinder.Eval(Container, "DataItem.Title")
```

то будет выдано сообщение об ошибке, которое гласит, что такого свойства отыскать не удалось. В данном случае вы располагаете единственной возможностью — преобразовать член в свойство, например, так:

```
public String Title
{
   get
   {
      return _title;
   }
   set
   {
      _title = value;
   }
}
```

Создание новых вхождений web-регистратора

На данный момент имеются все функциональные возможности, необходимые для извлечения и вывода списка вхождений, которые могут появиться на нашем сайте. Мы однако пока не умеем создавать новые вхождения без Notepad! В идеале желательно иметь на сайте форму, которую можно было бы использовать для создания новых вхождений.

Практикум: создание вхождений web-регистратора

1. Войдите в Solution Explorer, щелкните правой кнопкой мыши на проекте Weblog и выберите пункт меню Add | Add Web Form. Введите имя — `Edit.aspx`.

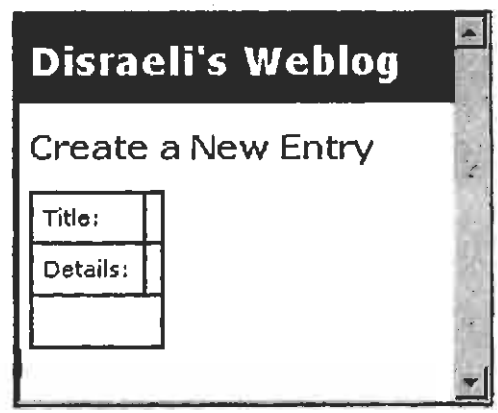
2. Откройте Designer для данной страницы. Щелкните мышью один раз на основание страницы и с помощью окна Properties замените значение свойства `pageLayout` на `FlowLayout`.

3. Выберите пункт меню View HTML Source. Добавьте следующий код:

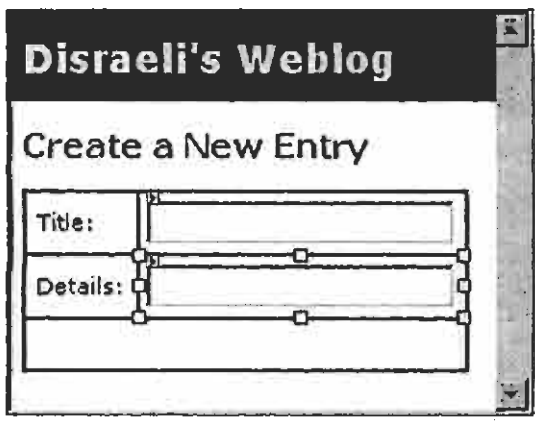
```
<%@ Page language="c#" Codebehind="Edit.aspx.cs" AutoEventWireup="false"
Inherits="WebLog.Edit" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
   <head>
      <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
      <meta name="CODE_LANGUAGE" Content="C#">
      <meta name="vs_defaultClientScript" content="JavaScript (ECMAScript)">
      <meta name="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5">
      <link rel="stylesheet" href="style.css">
   </head>
   <body>
      <form id="Edit" method="post" runat="server">
         <div class="header">
            Disraeli's Weblog
         </div>
         <br>
         <div class="normalHeading">
            Create a New Entry
         </div>
         <br>
         <div class="normal">
            <table cellspacing="0" cellpadding="3">
               <tr>
                  <td class="normal">
                     Title:
                  </td>
                  <td>
                  </td>
               </tr>
               <tr>
                  <td class="normal">
                     Details:
                  </td>
                  <td>
                  </td>
               </tr>
               <tr>
                  <td colspan="2" align="right">
```

```
                    </td>
                </tr>
            </table>
        </div>
    </form>
</body>
</html>
```

4. Выберите пункт меню View | Design; вы должны получить изображение, аналогичное изображению на рисуке справа.



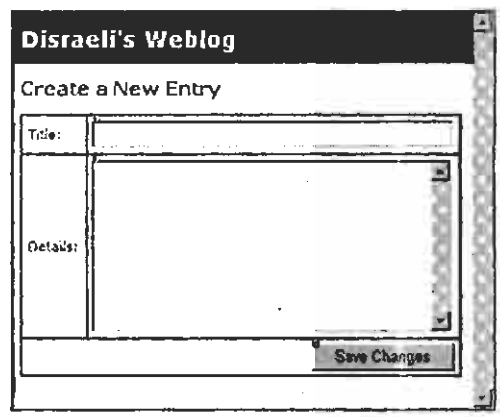
5. Воспользовавшись окном с инструментами, перенесите мышью управляющий элемент `Textbox` и разместите его в маленьком окошке, расположенном справа от `Title`.



Повторите операцию, но на этот раз разместите управляющий элемент в маленьком окошке, расположенном справа от **Details** (см. рис. слева).

- Присвойте свойству `ID` верхнего управляющего элемента значение `textTitle`.
- Присвойте свойству `ID` нижнего управляющего элемента значение `textTitle`. Измените значение его свойства `TextMode` на `Multiline`.

6. Воспользовавшись окном с инструментами, перенесите мышью управляющий элемент `Button` и разместите его в длинном пустом окне, расположенном в самом низу. Измените значение его свойства `ID` на `buttonOk`, а свойства `Text` — на `Save Changes` (сохранить изменения).



7. Слегка увеличьте размеры двух управляющих элементов `Textbox`, как показано на рисунке справа.

8. Мы хотим добиться того, чтобы при нажатии кнопки **Save Changes** пользователь сохранял изменения только тогда, когда оба текстовых окна заполнены. Для этого можно воспользоваться проверкой допустимости, предоставляемой ASP.NET, которая автоматически проверяет, что вся необходимая информация введена, а если нет, то выводит соответствующее сообщение.

9. Перенесите мышью управляющий элемент `RequiredFieldValidation` и разместите его справа от текстового окна **Title**. Измените значение его свойства `ErrorMessage` на `Required`, а свойству `ControlToValidate` присвойте значение `textTitle`.

10. Повторите действия, описанные в п. 9, но на этот раз перенесите мышью новый управляющий элемент и разместите его справа от окна **Details**. Измените значение его свойства `ErrorMessage` на `Required`, а свойству `ControlToValidate` присвойте значение `textDetails`.

11. Войдите в Solution Explorer и откройте редактор кода для `Entry`. Добавьте следующий член:

```
/// <summary>
///Краткое описание класса Entry.
/// </summary>
public class Entry
{
   // члены...
   private String _filename;
   private DateTime _timestamp;
   private String _title;
   private String _details;
```

12. Добавьте следующее свойство:

```
[XmlIgnore()] public String Filename
{
   get
   {
      return _filename;
   }
   set
   {
      _filename = value;
   }
}
```

13. Добавьте в класс следующий метод:

```
public void Save()
{
   // имеется ли у нас имя файла?
   if(Filename == null)
   {
      // получение имени файла на основе даты...
      Timestamp = DateTime.Now;
      Filename = String.Format("{0:d4}{1:d2}{2:d2}_{3:d2}{4:d2}.xml",
                               (int)Timestamp.Year, (int)Timestamp.Month,
                               (int)Timestamp.Day, (int)Timestamp.Hour,
                               (int)Timestamp.Minute);
   }

   // получение полного имени файла...
   String filepath = Global.EntryFilePath + "\\" + Filename;

   // создание сериализатора и сохранение...
   FileInfo fileInfo = new FileInfo(filepath);
   if(fileInfo.Exists == true)
      fileInfo.Delete();
   FileStream stream =
            new FileStream(fileInfo.FullName, FileMode.Create);
   XmlSerializer serializer = new XmlSerializer(this.GetType());
   serializer.Serialize(stream, this);
   stream.Close();
}
```

14. Откройте редактор кода для файла `Edit.aspx.cs`. Найдите метод `Page_Load()` и добавьте в него следующий код:

```
private void Page_Load(object sender, System.EventArgs e)
{
   // была ли сохранена данная страница?
   if(IsPostBack == true)
   {
      // создание объекта нового вхождения...
      Entry newEntry = new Entry();

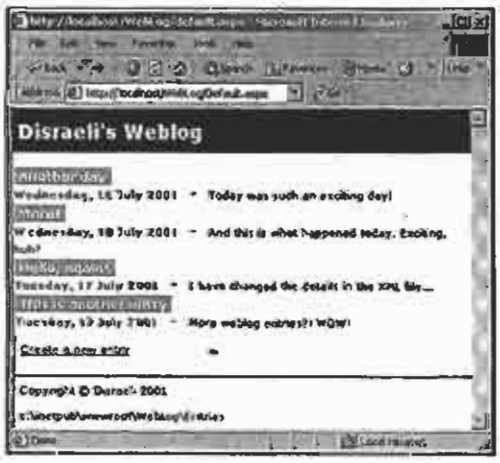
      // присваивание значений...
      newEntry.Title = textTitle.Text;
      newEntry.Details = textDetails.Text;

      // сохранение объекта...
      newEntry.Save();

      // вывод списка...
      Response.Redirect("default.aspx");
   }
}
```

15. Откройте редактор HTML для файла `Default.aspx`. Добавьте следующий код — часть кода, которую мы добавили к странице ранее, опущена для краткости:

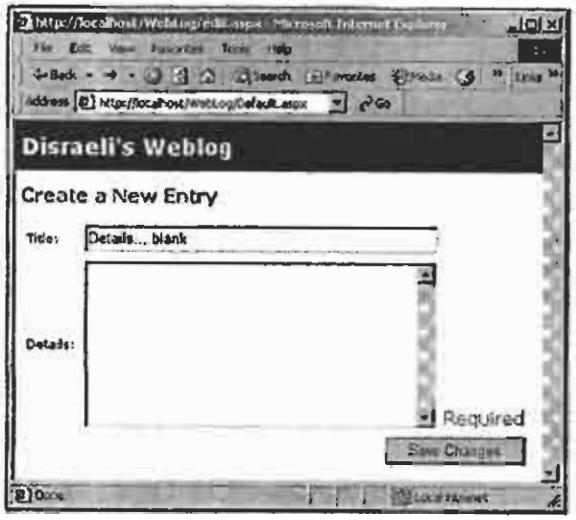
```
<asp:DataList id="datalistEntries" runat="server">
...
</asp:DataList>
<br>
<div class="normal">
   <a href="edit.aspx">Create a new entry</a>
</div>
<br>
<hr color="#000000">
<div class="normal">
   <asp:label id="labelCopyright" runat="server">(copyright)</asp:label>
</div>
```



Запустите проект. Щелкните на ссылке Create a new entry (создание нового вхождения). На экране появится форма. Заполните ее и щелкните на кнопке Save Changes. Страница перезагрузится, новое вхождение станет видимым (см. рис. слева).

В папке Entries создан новый файл (см. рис. справа).





Как это работает

Создание формы в `Edit.aspx` практически не отличается от создания страницы `Default.aspx`. Для обоих классов было использовано сочетание HTML-кода и редактора.

При нажатии кнопки на форме выполняется код, производящий проверку допустимости в соответствии с управляющими элементами, которые были добавлены на странице. Каждый из них был приписан отдельному управляющему элементу `Textbox`. При нажатии кнопки выполняется код, проверяющий все ли элементы заполнены, и если не все, то код выводит сообщение (см. рис. слева).

Если проверка допустимости пройдена успешно, то страница отправляется обратно ASP.NET. В этот момент снова вызывается метод `Page_Load()`, но есть возможность проверить и убедиться в том, что страница действительно посылалась обратно; для этих целей используется свойство `IsPostBack`:

```
private void Page_Load(object sender, System.EventArgs e)
{
   // была ли сохранена данная страница?
   if(IsPostBack == true)
   {
```

Если страница посылается обратно, то мы создаем новый объект `Entry`:

```
      // создание объекта нового вхождения...
      Entry newEntry = new Entry();
```

Когда страница посылается обратно, свойство `Text` управляющих элементов `Textbox` автоматически наполняется тем, что было введено пользователем. Это означает, что есть возможность поступать следующим образом:

```
      // присваивание значений...
      newEntry.Title = textTitle.Text;
      newEntry.Details = textDetails.Text;
```

Далее мы требуем от объекта `Entry`, чтобы он сохранил самого себя (ниже этот вопрос будет рассмотрен более подробно):

```
      // сохранение объекта...
      newEntry.Save();
```

После того как вхождение сохранено, мы обращаемся к ASP.NET, с тем чтобы браузер перешел к `Default.aspx` и можно было бы увидеть вновь созданное вхождение:

```
      // вывод списка...
      Response.Redirect("default.aspx");
   }
}
```

Для сохранения вхождения требуется имя файла, которое должно быть уникальным (в противном случае будут затерты другие вхождения), поэтому имя файла составляется с использованием даты и времени, когда вхождение было сохранено:

```
public void Save()
{
   // есть ли у нас имя файла?
   if(Filename == null)
   {
      // получение имени файла на основе даты...
      Timestamp = DateTime.Now;
      Filename = String.Format("{0:d4}{1:d2}{2:d2}_{3:d2}{4:d2}.xml",
                               (int)Timestamp.Year, (int)Timestamp.Month,
                               (int)Timestamp.Day, (int)Timestamp.Hour,
                               (int)Timestamp.Minute);
   }
```

Имя файла сохраняется в виде свойства (причина, по которой мы поступаем именно так, станет очевидной несколько позже), и для того чтобы в действительности сохранить его, необходимо преобразовать имя файла в полный физический путь:

```
   // получение полного имени файла...
   String filepath = Global.EntryFilePath + "\\" + Filename;
```

Получив это имя, мы используем объект `XmlSerializer` для сохранения объекта на диске. Метод сериализации получает в качестве объекта поток, представляющий собой файл, в который будет осуществляться запись, и объект `Entry`, после чего свойства последнего преобразовываются в XML-строку, т. е. осуществляются действия, прямо противоположные тем, которые перед этим выполнялись функцией `Deserialize()`:

```
   // создание сериализатора и сохранение...
   FileInfo fileInfo = new FileInfo(filepath);
   if(fileInfo.Exists == true)
      fileInfo.Delete();
   FileStream stream = new FileStream(fileInfo.FullName, FileMode.Create);
   XmlSerializer serializer = new XmlSerializer(this.GetType());
   serializer.Serialize(stream, this);
   stream.Close();
}
```

После того как файл `Default.aspx` будет перезагружен, метод `GetAllEntries()` вернет полный список файлов, включая и только что созданный.

Редактирование вхождений web-регистратора

Теперь у нас есть форма, которую можно использовать для создания новых вхождений. Но может ли она пригодиться для редактирования уже существующих XML-файлов на диске?

В этом разделе мы познакомимся с тем, каким образом нужно изменить приложение, чтобы редактировать уже имеющиеся вхождения.

Практикум: редактирование вхождений web-регистратора

1. Откройте Designer для `Default.aspx`.
2. Щелкните правой кнопкой мыши на управляющем элементе `DataList` и выберите пункт меню **Edit Template | Item Template**.
3. Воспользовавшись окном с инструментами, перенесите мышью управляющий элемент `Hyperlink` и разместите его рядом с управляющим элементом `Label` зеленого цвета (см. рис. на стр. 715).



4. Используя окно Properties, выберите свойство `DataBindings` управляющего элемента `Hyperlink`. Из списка, расположенного слева, выберите пункт NavigateUrl.

5. Щелкните мышью на переключателе Custom binding expression и введите следующее выражение:

```
"edit.aspx?filename=" + DataBinder.Eval(Container,
                                        "DataItem.Filename")
```

6. Щелкните мышью на OK чтобы сохранить изменения в привязке.

7. Снова воспользуйтесь окном Properties — измените значение свойства `Text` управляющего элемента `Hyperlink` на `Edit`, а значение свойства `CssClass` — на `entry`.

8. Откройте редактор кода для файла `Global.asax`. Найдите метод `LoadEntry()` и внесите в него следующие изменения:

```
// LoadEntry - загрузка вхождения с диска...
public static Entry LoadEntry(String filename)
{
   // У нас есть имя, теперь необходимо узнать путь...
   String filepath = EntryFilePath + "\\" + filename;

   // открытие файла...
   FileStream file = new FileStream(filepath, FileMode.Open);

   // создание сериализатора...
   XmlSerializer serializer = new XmlSerializer(typeof(Entry));
   Entry newEntry = (Entry)serializer.Deserialize(file);

   // закрытие файла...
   file.Close();

   // обновление времени, даты и имени файла...
   newEntry.Timestamp = new FileInfo(filepath).LastWriteTime;
   newEntry.Filename = filename;

   // возврат вхождения...
   return newEntry;
}
```

Запустите проект. Щелкните мышью на ссылке Edit, и вы увидите, что появится форма Create New Entry. Но самое интересное — это линейка Address. Имя XML-файла, в котором содержатся данные текущего вхождения, оказались "встроенными" в URL-адрес:



Как это работает

На данный момент мы пока еще не умеем редактировать нашу страницу, однако очередной задачей является внесение таких изменений в `Edit.aspx` и `Edit.asp.cs`, которые позволят нам найти имя файла соответствующего вхождения в URL-адресе и загрузить вхождение.

Самое интересное здесь — это привязка данных. Мы привязываем свойство NavigateUrl нового управляющего элемента Hyperlink к следующему выражению:

```
"edit.aspx?filename=" + DataBinder.Eval(Container, "DataItem.Filename")
```

Можно увидеть, каким образом — на каждой итерации по массиву объектов Entry — извлекается значение свойства Filename очередного объекта и присоединяется в конец строки edit.aspx?filename=. Это позволяет получить полный URL-адрес.

Загрузка выбранного вхождения

Чтобы загрузить выбранное вхождение в момент просмотра Edit.aspx, требуется всего лишь отыскать "параметр" filename, который может быть, а может и не быть включен в URL-адрес.

Практикум: загрузка выбранного вхождения

1. Откройте редактор кода для файла Edit.aspx.cs. Найдите метод Page_Load() и добавьте следующий код:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // сохранена ли страница?
    if(IsPostBack == true)
    {
        // создание объекта для нового вхождения...
        Entry newEntry = new Entry();

        // есть ли у нас имя файла, которое следует использовать?
        if(Request.Params["filename"] != null)
            newEntry.Filename = Request.Params["filename"];

        // присваивание значений...
        newEntry.Title = textTitle.Text;
        newEntry.Details = textDetails.Text;

        // сохранение вхождения...
        newEntry.Save();

        // вывод списка...
        Response.Redirect("default.aspx");
    }
    else
    {
        // удалось ли нам получить имя файла?
        String filename = Request.Params["filename"];
        if(filename != null)
        {
            // загрузка объекта вхождения...
            Entry entry = Global.LoadEntry(filename);

            // наполнение его полей...
            textTitle.Text = entry.Title;
            textDetails.Text = entry.Details;
        }
    }
}
```

Запустите проект. Теперь, если вы щелкните мышью на ссылке Edit, то вхождение будет загружено в новую страницу. Все изменения, внесенные в данные текущей формы, будут сохранены в XML-файле после нажатия Save Changes.

Как это работает

В .NET параметры могут поступать на страницу двумя способами: либо через переменные текущей формы, либо через строку запроса. В общем и целом, ASP.NET выполняет одну замечательную вещь — он позволяет скрыть то, каким именно образом формы работают в Паутине; все, о чем приходится беспокоиться, — это переменные строки запроса.

Переменные строки запроса — это перемеиные, которые располагаются в URL-адресе после вопросительного знака. В данном случае имеется единственная переменная с именем `filename`, которая обладает значением `20010718_1419.xml`:

```
http://localhost/WebLog/edit.aspx?filename=20010718_1419.xml
```

А вот следующий URL, напротив, включает в себя две переменные `a` и `b`; переменной `a` присваивается значение `Jack`, а переменной `b` — `Coffee`. Переменные отделяются одна от другой символом амперсанда (&):

```
http://anotherserver/page/aspx?a=Jack&b=Coffee
```

Если в метод `Page_Load()` возвращается значение `IsPostBack`, равное `false`, то это означает, что данная страница загружается в первый раз. Это предоставляет возможность присвоить в качестве начальных значений свойству `Text` информацию, которая хранится в файле (часть кода опущена для краткости):

```
private void Page_Load(object sender, System.EventArgs e)
{
   // сохранена ли страница?
   if(IsPostBack == true)
   {
      ...
   }
   else
   {
      // удалось ли получить имя файла?
      String filename = Request.Params["filename"];
      if(filename != null)
      {
```

Свойство `Request.Params`, выделенное в приведенном выше коде, может быть использовано для осуществления доступа к параметрам, которые передаются странице. В данном примере мы пытаемся получить и получаем значение параметра `filename`. Если бы мы щелкнули мышью на ссылке **Create new entry**, то этот параметр был бы пустым. Поскольку есть параметр, то мы осуществляем загрузку вхождения с диска и обновляем свойство `Text` каждого управляющего элемента:

```
         // загрузка объекта вхождения...
         Entry entry = Global.LoadEntry(filename);

         // наполнение его полей...
         textTitle.Text = entry.Title;
         textDetails.Text = entry.Details;
      }
   }
}
```

Вторая часть этой проблемы возникает при отправке страницы обратно. Снова приходится обращаться к свойству `Request.Params`, чтобы узнать, есть ли уже какое-либо имя файла. Если имеется, то именно его и следует использовать — в таком случае метод `Save()` заменит существующий файл на новый, содержащий все сделанные изменения. Если же имя отсутствует, то метод `Save()` воспользуется

текущими датой и временем для создания нового имени (здесь часть кода также опущена):

```
private void Page_Load(object sender, System.EventArgs e)
{
   // сохранена ли страница?
   if(IsPostBack == true)
   {
      // создание объекта для нового вхождения...
      Entry newEntry = new Entry();

      // есть ли у нас имя файла, которое следует использовать?
      if(Request.Params["filename"] != null)
         newEntry.Filename = Request.Params["filename"];

      // присваивание значений...
      newEntry.Title = textTitle.Text;
      newEntry.Details = textDetails.Text;

      // сохранение вхождения...
      newEntry.Save();

      // вывод списка...
      Response.Redirect("default.aspx");
   }
```

Мы иаучились выводить список вхождений и можем использовать web-сайт для создания новых вхождений и для внесения изменений в уже существующие.

Сортировка вхождений

Последнее созданное вхождение не выводится в самом верху страницы. Однако при традиционном способе работы web-регистраторов первым должно выводиться вхождение, введенное самым последним.

Поэтому требуется сортировать вхождения по дате при считывания их из директории. .NET возвращает файлы в алфавитном порядке, и именно поэтому мы используем уже созданный формат. Рассмотрим следующие три файловых имени:

```
20010718_1318.xml
20010719_1003.xml
20010720_1823.xml
```

В данном случае используется формат yyyymmdd_hhmm. Его недостаток заключается в том, что файлы, предназначенные для работы с сайтом, могут создаваться не чаще одного в минуту. В данном случае это не представляет серьезной проблемы, поскольку мы обычно должны создавать не более одного файла в день. Файлы с именами в таком формате при сортировке по алфавиту автоматически располагаются от самого раннего к самому позднему. Все, что необходимо сделать,— это изменить порядок файлов на обратный относительно того, в котором они хранятся на диске.

Практикум: сортировка вхождений

1. Создайте как минимум два вхождения, воспользовавшись web-инструментом. Уничтожьте существующие XML-файлы, поскольку их имена не отвечают только что обсуждавшемуся формату. Автор создал у себя на компьютере три файла (см. рис. на стр. 719, вверхний).

Файлы выводятся в алфавитном порядке, т. е. в порядке, прямо противоположном требуемому.



2. Откройте редактор кода для файла `Global.asax`. Найдите метод `LoadAllEntries` и внесите в него следующие два изменения:

```
// LoadAllEntries - осуществляет
// загрузку всех вхождений с диска...
public static Entry[] LoadAllEntries()
{
    // определение пути, по которому располагаются вхождения...
    DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

    // получение списка файлов...
    FileInfo[] files = entryFolder.GetFiles();

    // создание массива вхождений...
    Entry[] entries = new Entry[files.Length];

    // просмотр в цикле всех файлов и их загрузка...
    int index = files.Length - 1;
    foreach(FileInfo file in files)
    {
        entries[index] = LoadEntry(file.Name);
        index--;
    }

    // возврат списка...
    return entries;
}
```



Откомпилируйте и запустите проект. Файлы будут выводиться в правильном порядке, как это проиллюстрировано на рисунке слева.

Как это работает

Вся хитрость заключается в том, что массив заполняется в порядке, обратном тому, в котором файлы хранятся на диске. При создании массива известно, какого он должен быть размера, поскольку предполагается, что он должен в точности соответствовать размерам массива объектов `FileInfo`, возвращаемого методом `GetFiles()`.

```
// LoadAllEntries - осуществляет загрузку всех вхождений с диска...
public static Entry[] LoadAllEntries()
{
    // определение пути, по которому располагаются вхождения...
    DirectoryInfo entryFolder = new DirectoryInfo(EntryFilePath);

    // получение списка файлов...
    FileInfo[] files = entryFolder.GetFiles();

    // создание массива вхождений...
    Entry[] entries = new Entry[files.Length];
```

При прохождении в цикле по объектам FileInfo мы обращаемся к методу Global.LoadEntry с целью создания нового объекта Entry для каждого объекта FileInfo. Мы сохраняем этот объект в массиве на позиции, определяемой переменной index, которая изначально соответствует последнему элементу массива и уменьшается на каждой итерации на единицу до тех пор, пока не будет просмотрен весь массив files:

```
    // просмотр в цикле всех файлов и их загрузка...
    int index = files.Length - 1;
    foreach(FileInfo file in files)
    {
       entries[index] = LoadEntry(file.Name);
       index--;
    }

    // возврат списка...
    return entries;
}
```

Использование сеансов и куки

Нами создан web-сайт, на который может зайти любой пользователь и создать новые вхождения web-регистратора или отредактировать уже существующие. Это представляется не слишком разумным — в идеале только владелец сайта должен иметь возможность вносить в него изменения.

Поэтому теперь необходимо предусмотреть способ, посредством которого владелец сайта мог бы себя аутентифицировать. Сделав это, мы сможем при попытке пользователя внести в сайт изменения, осуществлять проверку его личности и в соответствии с этим разрешать или не разрешать делать изменения.

Сеансовые режимы позволяют web-серверу соотнестим набор данных с конкретным web-браузером. При таком сценарии браузер как бы "помечается" при своем первом обращении к данному сайту. Такая пометка является уникальной и при последующих обращениях используется в качестве ключа, "отпирающего" данные, относящиеся к сайту.

В качестве такой пометки браузера web-сервер использует куки. (Куки — это маленькое информационное сообщение, которое web-сервер посылает клиентской машине и которое считывается каждый раз при последующих посещениях сайта.) При первом обращении к странице сайта соответствующих куки еще не существует, поэтому web-сервер создает новый сеанс пользователя и переносит ID этого сеанса в куки перед тем, как отправить его браузеру. В программе ASP.NET устанавливаются значения различных свойств, дополняющих информацию о сеансе.

В сеансе можно хранить большое количество произвольной информации, но, поскольку на это расходуется память сервера, следует стремиться к тому, чтобы хранилось как можно меньше информации. Например, стоит ли хранить в сеансе развернутую информацию о пользователе, если можно ограничиться только относительно небольшой ID пользователя, а за более подробной информацией обращаться только тогда, когда в ней действительно возникает необходимость?

Ограничение доступа к web-сайту

В данном примере мы собираемся хранить в сеансе логическое значение, которое будет означать, разрешено или не разрешено данному пользователю редактировать информацию на сайте.

Практикум: запрет редактирования

1. Откройте редактор кода для файла Global.asax.
Найдите метод Session_Start() и добавьте в него следующий код:

```
protected void Session_Start(Object sender, EventArgs e)
{
    // конфигурирование сеанса...
    Session["canedit"] = false;
}
```

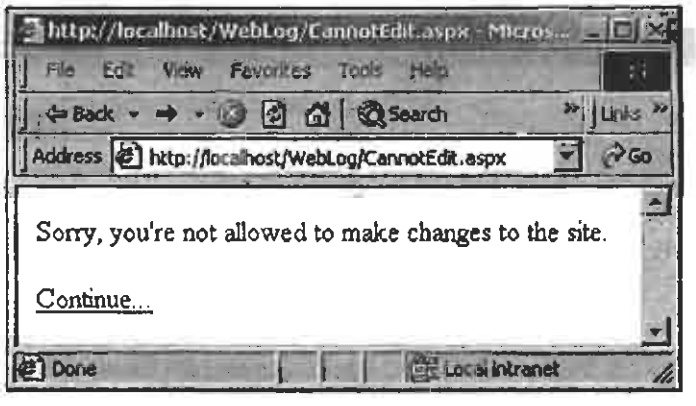
2. Откройте редактор кода для файла Edit.aspx.
Найдите метод Page_Load() и добавьте в него следующий код:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // разрешено ли нам выполнять редактирование...
    if((bool)Session["canedit"] == false)
      Response.Redirect("CannotEdit.aspx");
```

3. Воспользуйтесь Solution Explorer — щелкните правой кнопкой мыши на проекте Weblog и выберите пункт меню Add | Add Web Form. Создайте новую форму CannotEdit. Откройте редактор HTML для новой страницы и добавьте следующий код:

```
<%@ Page language="c#" Codebehind="CannotEdit.aspx.cs" AutoEventWireup="false"
Inherits="WebLog.CannotEdit" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
   <head>
      <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
      <meta name="CODE_LANGUAGE" Content="C#">
      <meta name="vs_defaultClientScript" content="JavaScript (ECMAScript)">
      <meta name="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5">
   </head>
   <body MS_POSITIONING="GridLayout">
      <form id="CannotEdit" method="post" runat="server">
         Sorry, you're not allowed to make changes to the site.
         /Простите, вы не можете вносить изменения в этот сайт./
         <br>
         <br>
         <a href="default.aspx">Continue...</a>
      </form>
   </body>
</html>
```

Откомпилируйте и запустите проект. Вы должны сразу заметить, что ссылки Edit и Create new entry приводят на страницу с сообщением об ошибке, а не на ту страницу, к которой мы имели доступ ранее (см. рис. справа).



Как это работает

Вся хитрость здесь заключается в том, что все это выполняет нечто, называемое сеансом. Сеанс — это средство ASP.NET, которое позволяет проводить идентификацию пользователя. Когда пользователь

запрашивает с сайта ASP или ASP.NET страницу в первый раз, то для данного пользователя создается сеанс, являющийся уникальным,— ни у кого другого не может быть точно такого же сеанса.

При создании нового сеанса, т. е. когда новый пользователь запрашивает первую страницу сайта, вызывается метод `Session_Start()`. И `System.Web.HttpApplication`, и `System.Web.UI.Page` поддерживают свойство `Session`. Это свойство представляет собой базовое семейство для хранения значений. В данном случае было создано значение с именем `canedit`, для которого было сохранено значение `false`:

```
protected void Session_Start(Object sender, EventArgs e)
{
   // конфигурирование сеанса...
   Session["canedit"] = false;
}
```

Такие значения называются переменными сеанса.

Мы обладаем доступом к этому значению из любой страницы сайта. При осуществлении доступа к переменным сеанса мы имеем гарантию того, что значения, которые мы получаем, *уникальны* для данного пользователя. Другими словами, если сто пользователей одновременно работают с сайтом, то у каждого из них имеется свой собственный уникальный сеанс. При создании сеанса переменной `canedit` в качестве начального значения присваивается `false`. Но если изменить значение этой переменной на `true` только для одного из этих пользователей, то это будет означать, что изменение коснется только того пользователя, для которого оно сделано. Мы хотим изменить значение переменной `canedit` на `true` только для того пользователя, который сможет подтвердить пароль.

При загрузке страницы `Edit.aspx.cs` производится проверка значения этой переменной, и если оно равно `false`, то пользователь переиаправляется на другую страницу.

```
private void Page_Load(object sender, System.EventArgs e)
{
   // разрешено ли нам выполнять редактирование...
   if((bool)Session["canedit"] == false)
      Response.Redirect("CannotEdit.aspx");
```

Аутентификация пользователя

В данном упражнении мы собираемся создать очень простую программу аутентификации. Ее действия, в основном будут заключаться в том, чтобы запросить у пользователя пароль и, при его совпадении с имеющимся у нас именем, присвоить переменной `canedit` значение `true`. В результате у данного конкретного пользователя появится возможность редактировать страницы.

Практикум: аутентификация пользователя

1. Войдите в Solution Explorer, щелкните правой кнопкой мыши на проекте Weblog и выберите пункт меню Add | Add Web Form. Назовите файл `Login.aspx`.

2. Войдите в Designer для данной формы, Щелкните мышью на основании страницы и с помощью окна Properties замените `pageLayout` на `FlowLayout`.

3. Мы не станем утруждать себя красивым оформлением этой страницы, поскольку она является инструментом администрирования и, кроме того, вы наверняка получили

общее представление о том, каким образом можно сделать страницу более привлекательной. Вместо этого перенесите на страницу новые управляющие элементы — `Textbox` и `Button`.

4. Выберите управляющий элемент `Textbox` и измените значение его свойства `ID` на `textPassword`. Измените значение его свойства `TextMode` на `Password`.

5. Выберите управляющий элемент `Button` и измените значение его свойства `Text` на `Login`.

6. Щелкните два раза мышью на основании страницы, для того чтобы открыть метод `Page_Load()`. Добавьте в него следующий код:

```
private void Page_Load(object sender, System.EventArgs e)
{
   // отправляем обратно?
   if(IsPostBack == true)
   {
      // правильно ли введен пароль?
      if(textPassword.Text.CompareTo("stringy") == 0)
      {
         // обновление сеанса...
         Session["canedit"] = true;

         // перенаправление...
         Response.Redirect("default.aspx");
      }
   }
}
```



Запустите проект. Щелкните мышью на ссылке Edit — вы, безусловно, должны быть возвращены на начальную страницу. Замените `Default.aspx` на `Login.aspx` в линейке Address и щелкните на кнопке Go (см. рис. слева).

В соответствующее поле введите в качестве пароля текст `stringy` (обязательно соблюдайте регистр). Щелкните на кнопке Login — в результате вы сразу будете перенесены на страницу `Default.aspx`. Теперь, если щелкнуть на ссылке Edit, то можно вносить изменения.

Как это работает

Каждый раз при загрузке странницы `Edit.aspx` программа осуществляет проверку — имеет или нет сеансовая переменная `canedit` значение `true`. Если нет, то мы будем перенаправлены на страницу с сообщением об ошибке.

Когда на странице `LoadXml()` нажимается кнопка Login, то происходит вызов метода `Page_Load()`. В этот момент производится проверка, посылалась ли страница обратно, а затем проверяется значение свойства `Text` управляющего элемента `textPassword` на совпадение введенного пароля со строкой `stringy`:

```
private void Page_Load(object sender, System.EventArgs e)
{
   // отправляем обратно?
   if(IsPostBack == true)
   {
      // правильно ли введен пароль?
      if(textPassword.Text.CompareTo("stringy") == 0)
      {
```

Если пароль совпадает, то мы изменяем значение сеансовой переменной `canedit` на `true` и вновь отправляем пользователя на первоначальную страницу.

```
            // обновление сеанса...
            Session["canedit"] = true;

            // перенаправление...
            Response.Redirect("default.aspx");
        }
    }
}
```

Теперь, когда бы мы ни нажали на соответствующую ссылку и ни открыли страницу `Edit.aspx`, проверка покажет, что переменная `canedit` имеет значение `true`, и, следовательно, страница будет выводиться обычным образом. В этом случае можно вносить все необходимые изменения.

Итоги

В этой главе мы познакомились с Active Server Pages .NET или ASP.NET. Это технология следующего поколения от компании Microsoft, которая таким образом развивает успех своей предшествующей технологии — ASP, появившейся в 1996 г.

Сначала мы рассмотрели вопрос о том, как в ASP.NET работает модель управляющий элемент/событие и узнали, что эта парадигма очень похожа на ту, что используется в Windows Forms. На самом деле ASP.NET оказывается очень простым для освоения новичками. Напротив, опытные web-разработчики столкнутся с трудностями, поскольку для них парадигма окажется совершенно новой.

Для того чтобы проиллюстрировать некоторые из приемов, используемых при создании приложений ASP.NET, нами был создан web-сайт, на котором можно просматривать вхождения, имеющиеся в web-регистраторе. Эти вхождения хранятся в виде XML-файлов и могут считываться и записываться с помощью класса `XmlSerializer` — чрезвычайно полезного класса .NET Framework. Нами были также созданы статические методы в классе `Global`, доступные всем страницам сайта и позволяющие извлекать отдельные объекты `Entry` или массив всех объектов `Entry`, расположенных на диске. Мы воспользовались управляющим элементом `DataList`, а также другими управляющими элементами, встроенными в ASP.NET, для вывода перечня объектов `Entry`.

Наконец, мы рассмотрели вопрос о том, каким образом можно создать форму, которая позволяла бы пользователю создавать и редактировать новые объекты `Entry` непосредственно на самом сайте. Мы также познакомились с семейством `Session` и получили некоторую информацию о состоянии сеанса.

Упражнения

1. Какой основной принцип лежит в основе ASP.NET?
2. Какое количество файлов в Visual Studio.NET привязывается к `.aspx`-файлу?
3. Какие действия позволяет выполнять управляющий элемент `DataList`?
4. Что такое таблица стилей?
5. Какие особенности отличают формат, который использовался при создании имен XML-файлов?

25

Web-службы

Возможно, термин "web-служба" вам уже встречался ранее, хотя вы можете и не знать, что из себя представляют эти службы и каким образом они подстраиваются под работу Паутины в настоящее время. Web-службы являются основой нового поколения web-приложений. Что бы ни представляло из себя клиентское приложение, будь это приложение Windows или приложение ASP.NET Web Forms, и какая бы система ни использовалась на клиентской машине, например, Windows, Pocket Windows или какая-либо другая, они будут постоянно взаимодействовать с Интернетом посредством web-службы. Web-службы напоминают компоненты, с которыми мы познакомились в главе 21, за исключением того, что они выполняются через Интернет. Это выполняющиеся на стороне сервера программы ожидают прихода сообщений от клиентских приложений и возвращают им какую-то специфическую информацию. Эта информация может исходить непосредственно от самой web-службы, от других компонентов, находящихся в том же самом домене, или от других web-служб. Хотя полная концепция web-служб и претерпела некоторые изменения на протяжении того периода, пока писалась эта книга, на самом деле, можно выделить несколько типов web-служб, которые предназначены для выполнения различных функций: некоторые web-службы предоставляют информацию о какой-то конкретной области, например, о производстве или здравоохранении; существуют портальные службы, которые используют службы различных провайдеров для предоставления информации по какой-либо конкретной теме; есть службы, предназначенные для работы с каким-либо одним типом приложений, и отдельные составные части служб, которые могут использоваться многими разнообразными приложениями.

Web-службы предоставляют возможность комбинировать, совместно использовать, обмениваться или подключаться к отдельным службам различных производителей и разработчиков с целью создания абсолютно новых служб или создаваемых по заказу приложений, которые формируются в динамике для удовлетворения нужд клиентов.

В настоящей главе будут рассматриваться следующие темы:

- Предшественники web-служб
- Что такое web-служба?
- Протоколы, используемые web-службами
- Создание web-службы ASP.NET
- Тестирование web-служб
- Создание клиента для пользования web-службами

Мы не будем углубляться в детали внутреннего устройства web-служб, в особенности служб, работающих в форматах SOAP и WSDL, которые основываются на использовании XML, однако вы сможете получить общее представление о том, для чего эти протоколы используются. Прочитав эту главу, вы сможете приступить к созданию и использованию простых web-служб с помощью Visual Studio.NET.

До того как появились web-службы

Объединение компьютеров для обмена данными представлялось важной концепцией уже 1969 г., когда из четырех компьютеров, соединенных телефонной линией, была создана первая сеть ARPANET. В 1976 г. был разработан протокол TCP/IP. Для упрощения этого протокола в университете Беркли была создана модель разъема для сетевого программирования.

При программировании с использованием **Sockets API** клиент должен был инициировать установление соединения с сервером, после чего он получал возможность отправлять и принимать данные. Чтобы добиться выполнения на сервере каких-либо операций и получить некоторые результаты, требуются дополнительные протоколы, необходимые для описания кодов запросов и ответов. Примерами подобных так называемых "протоколов приложений" могут служить FTP, Telnet и HTTP. Протокол FTP используется для получения от сервера и отправки на сервер различных файлов. Протокол FTP поддерживает коды запросов GET и PUT, которые пересылаются по соединению от клиента к серверу. Сервер анализирует поток получаемой информации и в соответствии с кодами полученных запросов вызывает соответствующие методы. Протокол HTTP работает практически так же, как и FTP (см. главу 23).

Вызов удаленной процедуры

Используя Sockets API и протокол TCP/IP для вызова методов на сервере, программисту пришлось бы создавать средства, с помощью которых сервер мог бы анализировать поток данных и вызывать соответствующий метод. Для упрощения ситуации целым рядом компаний был создан протокол RPC (Remote Procedure Calls — вызов удаленных процедур), одна из разновидностей которого до сих пор весьма популярна. Речь идет о протоколе **DCE-RPC** (Distributed Computer Environment — Remote Procedure Calls, распределенная компьютерная среда — вызов удаленных процедур), созданном компанией Open Software Foundation, которая позднее была преобразована в Open Group (см. www.opengroup.org). При использовании RPC мы определяем методы в формате IDL (Interface Definition Language — язык описания интерфейсов), которые должны быть выполнены сервером и которые могут быть вызваны клиентом. Эта работа выполняется специальной программой, которая называется "заглушкой" (stub) и генерируется компилятором интерфейса.

RPC предназначается для вызова методов, а это означает, что вам придется заниматься процедурным программированием. Технология RPC появилась относительно поздно, когда большинство разработчиков начали работать с парадигмой объектно-ориентированного программирования. Чтобы заполнить этот технологический пробел, было создано несколько различных технологий, включая CORBA и DCOM.

CORBA

Компания Object Management Group (OMG, www.omg.org) выступила инициатором создания **CORBA** (Common Object Request Broker Architecture — Единая архитектура брокера объектных запросов) в 1991 г. для придания "объектной ориентированности" сетевому программированию. Многие компании, такие как Digital Equipment, HP, IBM и др., разработали CORBA-серверы. Но поскольку компанией OMG были определены не конкретные требования по реализации, а только спецификации, то серверы различных фирм реально оказались несовместимыми. Серверу HP требовался клиент HP, серверу IBM был необходим клиент IBM и т. д.

DCOM

Совместно с Windows NT 4, компания Microsoft стала распространять протокол DCE-RPC с возможностями объектно-ориентированного программирования. Протокол DCOM (Distributed COM — распределенный COM) сделал возможным вызов COM-компонентов по сети; он используется в приложениях COM+. После нескольких лет, на протяжении которых обязательным требованием для использования DCOM являлось примененне операционных систем Microsoft, эта компания сделала свой протокол открытым для других посредством The Active Group (активная группа). DCOM стал доступен для Unix, VMS и операционных систем IBM. DCOM интенсивно использовался в различных средах компании Microsoft, однако попытка перенести его на другие системы по большому счету успеха не имела. Вряд ли администратор мейнфрейма IBM захочет включать в свою систему технологню компанин Microsoft.

RMI

Компания Sun выбрала свой путь с использованием собственных Java-технологий. В чистой Java-среде протокол **RMI** (Remote Method Invokation — вызов удаленных методов) использовался для вызова удаленных объектов. Компанией Sun были созданы определенные мосты для связи с CORBA и COM, но основной задачей компаннн было добиться массового использования решенний, основанных только на Java.

SOAP

Все технологнн, с которыми мы сталкивались до сих пор, использовались для взанмодействня между прнложеннями. Но если у вас есть решення на CORBA, DCOM и RMI, то очень сложно заставить их обменнваться ннформацией между собой. Другой недостаток, присущнй этим технологиям, заключается в том, что их архитектура не позволяет достичь масштабируемости при тысячах клиентов, что является обязательным требованием для Интернет-решений.

По этой прнчнне несколько компаннй, включая Microsoft и UserLand Software (www.userland.com), создали в 1999 г. **SOAP (Simple Object Access Protocol — простой протокол доступа к объектам)** как совершенно новый и абсолютно новаторскнй способ вызова объектов через Интернет, который строится на уже принятых стандартных протоколах. В SOAP для описания методов, подлежащих удаленным вызовам через сеть, и их параметров используется формат, основанный на XML. SOAP-сервер, работающий в среде COM, транслирует SOAP-сообщення в COM-вызовы, а SOAP-сервер, работающий в среде CORBA, транслирует их в CORBA-вызовы. Изначально для определения самого SOAP использовался протокол HTTP, поэтому SOAP-вызовы могут осуществляться через Интернет.

С появлением SOAP родился и термин "**web-служба**".

Спецификации W3C по SOAP 1.1 можно найти по адресу www.w3.org/TR/SOAP/.

В современном мире **каждое приложение, которое предоставляет какие-либо услуги, является web-службой**. Нет никакой необходимости, чтобы web-службой в качестве формата для сообщений обязательно использовался бы протокол SOAP, а для а для обмена информацией между клиентом и сервером — протокол HTTP; вместо них могут применяться другие протоколы. В спецификациях SOAP 1.0 использование HTTP выступало как обязательное требование, однако в версию 1.1 были внесены изменения, которые позволяют применять и другие транспортные протоколы. Несмотря на это, при использовании web-служб через Интернет протоколы SOAP и HTTP являются основными. С появлением .NET Remoting стало возможным создавать web-службы, которые используют сообщения в двоичном формате и TCP-соединение.

В настоящей главе основное внимание будет сосредоточено на web-службах, которые могут быть созданы с использованием Visual Studio.NET Project Wizard (мастера проектов Visual Studio.NET), который, в свою очередь, использует .NET Remoting, SOAP и HTTP.

Где находят применение web-службы

Чтобы взглянуть на web-службы под другим углом зрения, мы разделим их на обеспечивающие **взаимодействие пользователь—приложение** и на обеспечивающие **взаимодействие приложение—приложение**.

Начнем со **взаимодействия пользователь—приложение** и попробуем узнать из Паутины информацию о погоде. В сети существует огромное количество сайтов вроде http://www.msnbc.com и http://www.weather.com, в которых информация о погоде представлена в удобном для восприятия виде. Поэтому такие страницы обычно читаются непосредственно пользователем.

Если же мы хотим создать клиентское приложение с большими возможностями (**взаимодействие приложение—приложение**) для вывода информации о погоде, то нашему приложению будет необходимо установить соединение с соответствующим web-сайтом посредством URL-строки, в которой бы содержалось название города, погоду в котором мы хотим узнать. После этого нам пришлось бы проанализировать полученное в результате HTML-сообщение, пришедшее от web-сайта, для того чтобы получить значения температуры и других погодных условий, после чего мы смогли бы, наконец, вывести эту информацию в удобном формате клиентскому приложению с большими возможностями.

Это довольно большая работа особенно потому, что требуется получить по несколько температурных сводок для каждого конкретного города. Более того, процесс извлечения данных из HTML оказывается совсем не тривиальным. Ситуация складывается так потому, что HTML-данные обладают структурой, пригодной для вывода в web-браузере, и не предназначены для использования каким-либо бизнес-приложением со стороны клиента. Соответственно, все данные оказываются встроенными в текст, и извлечь их оказывается не так просто. Вам придется переписывать заново или подстраивать клиентское приложение так, чтобы оно было в состоянии извлекать разнородную информацию (например, о ливне) с одной и той же web-страницы. Напротив, при использовании web-браузера пользователи обладают возможностью немедленно получить интересующие их данные.

Чтобы избавиться от проблем с обработкой HTML-данных, web-служба может предоставлять некоторые полезные средства, которые возвращают только запрошенные данные. Надо просто вызвать метод на удаленном сервере и получить необходимую информацию, которая может непосредственно использоваться клиентским приложением. Ни на каком этапе не приходится иметь дела с предварительно отформатированным текстом, предназначенным для пользовательского интерфейса, поскольку web-служба предоставляет информацию в XML-формате, а инструменты для обработки XML-данных давно разработаны. Все, что требуется от клиентского приложения,— это вызвать некоторые методы XML-классов .NET Framework, которые позволяют получить требуемую информацию. Если мы пишем программу-клиент для web-службы .NET на C#, то вообще не нужно писать для этого никакого кода, поскольку существуют инструменты, которые создадут код на C# за нас!

Приложение для сообщений о погоде является только одним из возможных применений web-служб. Кроме него существует огромное множество других.

Сценарий приложения для организации путешествий

Как вы организуете свой отдых в праздничные дни? Вы идете в бюро путешествий, которое выполняет вместо вас всю необходимую работу. А организовывали ли вы когда-нибудь свой отдых через Интернет? На web-сайте авиакомпании вы можете ознакомиться с расписанием полетов и купить подходящий билет. Программа поиска по web-сайтам может быть использована для выбора отеля в соответствующем городе. Возможно, вам повезет, и вы сумеете найти карту города, на которой будет указана дорога к выбранному отелю. Отыскав страничку отеля, вы можете перейти к форме, посредством которой осуществляется бронирование мест. После этого вы получаете возможность перейти к поиску фирмы, осуществляющей прокат автомобилей...

Большая часть работы, которую вам пришлось выполнять, заключалась в поиске нужных web-сайтов с помощью программ поиска и в определении путей, по которым к этим сайтам можно добраться. Вместо этого мы могли бы создать **приложение для организации путешествий**, которое использовало бы различные web-службы, содержащие подробную информацию об отелях, воздушном сообщении, компаниям по прокату автомобилей и т. д. и представляло бы клиенту простой в использовании интерфейс, который позволял бы ему решать все проблемы, связанные с проведением отдыха, включая музыкальное напоминание о предварительном заказе. Используя свой карманный PC в месте проведения отдыха, пользователь мог бы использовать ту же самую web-службу для получения карты места отдыха, для получения актуальной информации о программах кинотеатров и т. д.

Сценарий приложения дистрибьютера книжной продукции

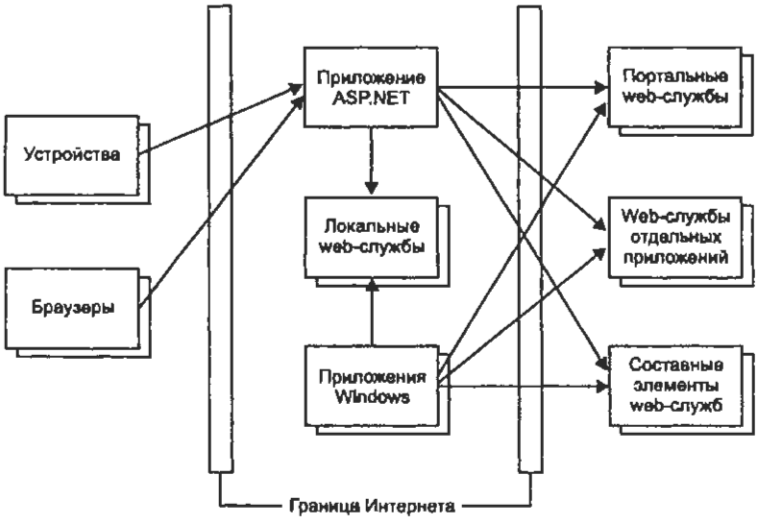
Web-службы могут также оказаться полезными для двух компаний, связанных между собой некоторыми партнерскими отношениями. Предположим, что дистрибьютер книжной продукции заинтересован в распространении информации по книжным магазинам об имеющихся у него на складе книгам. Это может быть реализовано в виде web-службы. Можно создать специальное приложение ASP.NET, использующее такую web-службу, которое предлагало бы пользователям непосредственный доступ к этой web-службе. Другое клиентское приложение такой web-службы могло бы представлять собой приложение Windows для книжного магазина, которое сначала осуществляет ревизию своих складов, а затем проверяет наличие книг на складе дистрибутора. Продавец может дать немедленный ответ относительно дат поставок без необходимости проверять какие-либо склады с помощью других приложений.

Типы клиентских приложений

Клиентом web-службы может быть либо приложение Windows, созданное с использованием Windows Forms, либо приложение ASP.NET, созданное с использованием Web Forms. Стать потребителем web-службы (использовать ее) могут Windows PC, система Unix или карманный PC. При использовании .NET Framework web-службы могут использоваться любым типом приложений: Windows Forms, Web Forms или консольным приложением.

Архитектура приложения

Каким образом должно выглядеть приложение, которое является потребителем web-службы? Независимо от того, разрабатываете ли вы приложение ASP.NET, приложение Windows или приложение для малых устройств, web-службы оказываются важной технологией для всех типов приложений.



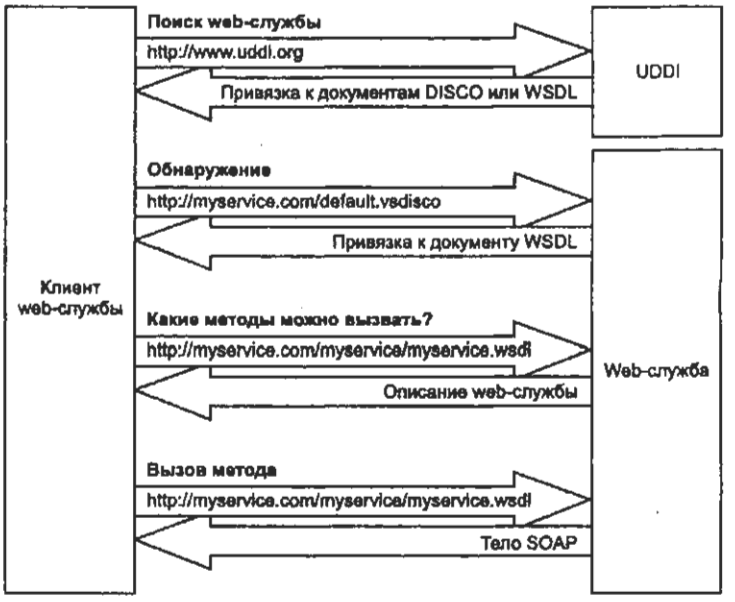
На рисунке слева изображен сценарий возможного использования web-служб. Различные устройства и браузеры соединены через Интернет с приложением ASP.NET, которое разработано с применением Web Forms. Приложение ASP.NET использует определенные локальные web-службы, а также те удаленные web-службы, доступ к которым может быть осуществлен через сеть: портальные web-службы, web-службы конкретного приложения и составные web-службы. Следующий перечень поможет разобраться в предназначении каждого из этих типов служб:

- **Портальные web-службы** предлагают услуги от различных компаний, занимающихся одним и тем же. Это простая в использовании единая точка доступа к нескольким службам.
- **Web-службы конкретного приложения** создаются для использования единственным приложением.
- **Составные части web-служб** представляют собой службы, которые с легкостью могут быть использованы многими приложениями. Разработка **Hailstorm** компании Microsoft предлагает большое количество подобных услуг, например, сохранение конфигурации приложения или аутентификацию пользователей.

Приложение Windows, изображенное на рисунке, может использовать web-службы напрямую, минуя приложение ASP.NET.

Архитектура web-служб

Если мы планируем воспользоваться уже существующей web-службой, то нам необходимо найти web-службу, которая удовлетворяет нашим требованиям. Если у нас есть информация о существовании подходящей web-службы, то нам потребуется информация о том, как можно к ней обратиться. Основные механизмы вызова web-служб изображены на рисунке справа.



- Прежде всего необходимо найти web-службу, уже зарегистрированную в службе директории регистрации, которая возвращает информацию, необходимую для привязки к Web-службе.
- Следующее действие — **обнаружение**. Если известен сервер, на котором располагается web-служба, то можно затребовать у сервера описание этой web-службы, для чего используется протокол обнаружения (DISCO).
- **Описание** службы представляется на языке Web Services Description Language (WSDL — язык описания web-служб). В таком описании содержится информация о том, какими методами располагает web-служба и какие типы аргументов им необходимо передавать. При использовании службы, которая не поддерживает обнаружение и не регистрируется в директории UDDI, все, что мы получим,— это документ WSDL.
- Располагая описанием web-службы, мы обладаем информацией о том, какие методы можно вызвать. Методы будут вызываться с помощью SOAP, поэтому все обращения к методам, включая все аргументы, должны быть преобразованы в соответствии с протоколом SOAP.

Как SOAP, так и WSDL описываются посредством XML-грамматики.

Рассмотрим подробнее каждый из этих последовательных шагов.

Программа поиска для web-служб

Существует определенная вероятность того, что вы сможете воспользоваться web-службой, которая уже поддерживается некоторой другой компанией. Для того чтобы иметь возможность осуществлять поиск и находить заранее существующие web-службы, компании Microsoft, IBM и Ariba объединились и организовали web-сайт www.uddi.org, на котором располагается служба **UDDI** (Uniform Description, Discovery and Integration — **единое описание, обнаружение и объединение**).

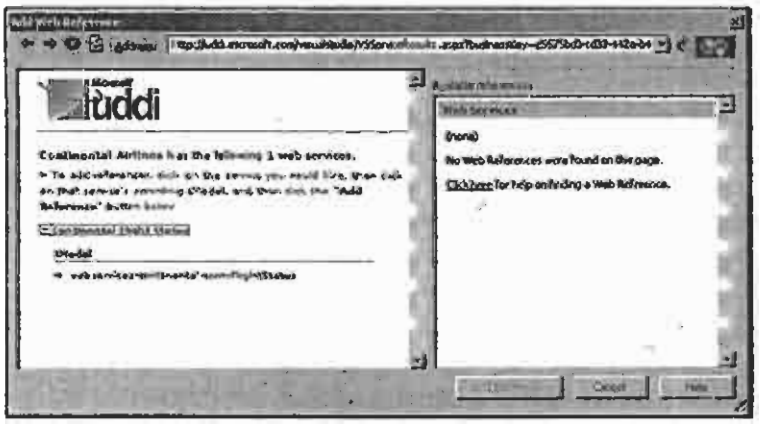
Служба UDDI является не зависящей от используемой платформы открытой системой для описания служб, обнаружения коммерческих объектов и объединения коммерческих служб с помощью Паутины, и, кроме того, она выполняет функции рабочего реестра. Любая компания, которая желает объявить о своей службе, может ее здесь зарегистрировать. С помощью коммерческого реестра UDDI и UDDI API можно заносить информацию о web-службах программным путем.

После инициализации проекта UDDI тремя компаниями его поддерживают более 220 компаний. Среди них Boeing, BT, Compaq, DataChannel, Dell, Fujitsu, HP, Hitachi, KPMG Consulting, Merrill Lynch, Nortel Networks, Oracle, Rational, SAP, Sun Microsystems, VeriSign и многие другие.

Для выявления уже существующей web-службы, можно осуществить поиск по коммерческому имени или путем просмотра, воспользовавшись одной из следующих категорий: промышленная классификация, географическая классификация ISO 3166 или географическая классификация компании Microsoft. После успешного завершения поиска службы, вы получаете описание этой web-службы; полную информацию, касающуюся классификации этой web-службы, где описываются группы, к которым она принадлежит; информацию о привязке, куда включается файл обнаружения (DISCO) и, возможно, документ WSDL.

В Visual Studio.NET поиск по сайту UDDI компании Microsoft может быть осуществлен с помощью меню Project | Add Web Reference.

Кроме **Microsoft UDDI Directory** существует еще **Test Microsoft UDDI Directory**. На период разработки собственной web-службы вы можете зарегистрировать вашу службу в тестовой директории, вторую директорию следует использовать только для отлаженных активных служб.



Если мы войдем в Microsoft UDDI Directory, то сможем задавать строки, по которым будет вестись поиск зарегистрированных web-служб. Если же в результате такого поиска будет получена информация о привязке, то можно включить ссылку на web-службу с помощью кнопки Add Reference.

Если задать строку `Continental`, то будет выведена web-служба компании Continental Airlines — Continental Flight Status, как показано на рисунке слева.

На момент написания этой книги на сайте UDDI компании Microsoft было зарегистрировано не так уж много web-служб, а у многих из зарегистрированных были нефункционирующие ссылки. Эта ситуация, скорее всего, изменится с появлением Visual Studio.NET.

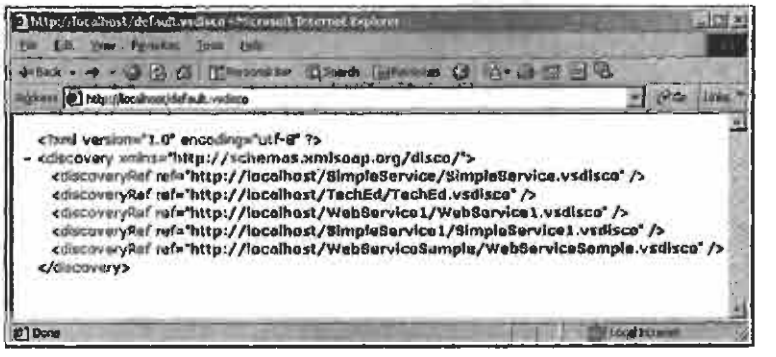
Большое количество web-служб можно найти на страницах www.xmethods.net и www.gotdotnet.com.

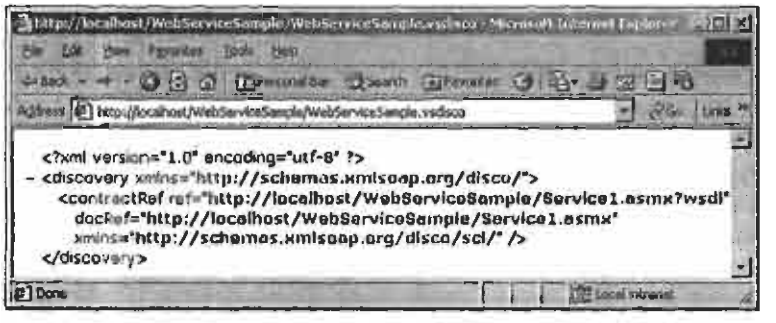
Обнаружение web-служб

Если вы знаете, что некий сервер поддерживает web-службы, то для получения информации о том, какие именно web-службы им поддерживаются, следует использовать процедуру **обнаружения**. Для хранения этой информации web-сервер использует файл с расширением `vsdisco`.

В файле vsdisco определяются правила, которыми сервер руководствуется при поиске web-служб. С помощью этого файла определяется, что службы должны разыскиваться либо динамически с исключением некоторых директорий, либо статически с указанием директорий, в которых должен осуществляться поиск.

Запросив с помощью браузера файл Default.vsdisco, можно получить ссылки на остальные файлы с расширением vsdisco, содержащиеся в элементе discoveryRef. Один из элементов <discoveryRef> содержит ссылку на http://localhost/WebServiceSample/WebServiceSample.vsdisco. Мы воспользуемся этой ссылкой для получения дополнительной информации об этой службе (см. рис. справа).





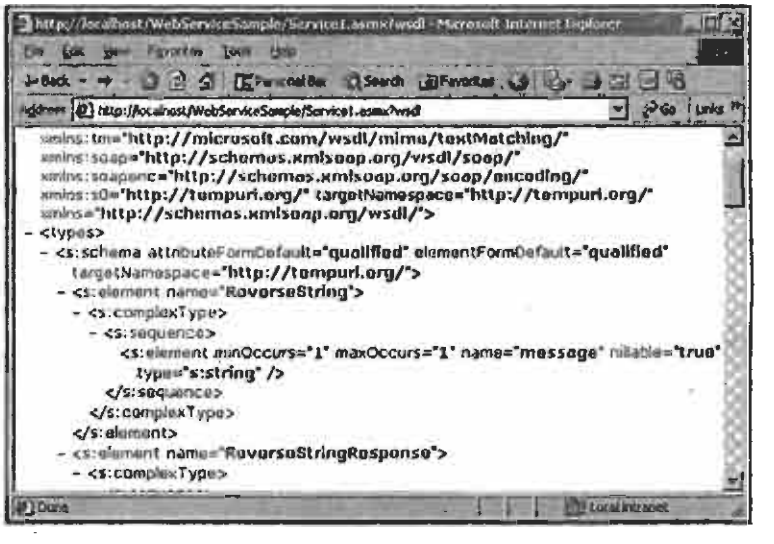
Если в адресное окно браузера ввести адрес http://localhost/WebServiceSample/WebServiceSample.vsdisco, то браузер вернет ссылку на WSDL-информацию относительно данной службы с помощью элемента <contractRef>: http://localhost/WebServiceSample/Service1.asmx?wsdl, как показано на рисунке слева.

Использовать браузер для чтения информации совершенно не обязательно. Информация для обнаружения может использоваться непосредственно инструментами Visual Studio.NET: в диалоге Add Web Reference вы можете обнаружить гиперссылку Web References on Local Web Server, где располагается информация локального сервера, которая используется при поиске web-служб. Также есть возможность ввести адрес web-службы в окно редактирования Address.

Какие методы можно вызвать?

Документ, составленный на языке описания web-служб (Web Services Description Language, WSDL), содержит в себе всю информацию о методах, которые поддерживаются данной web-службой, и о том, как они могут быть вызваны: типы параметров, передаваемых Web-службе, и типы возвращаемых web-службой значений.

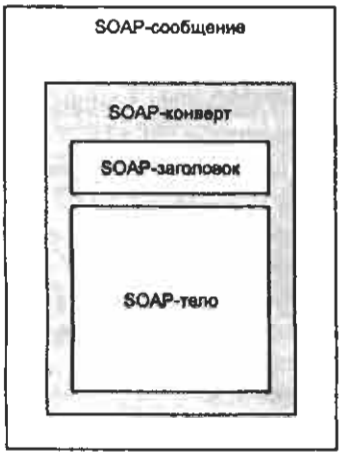
Воспользовавшись элементом <contractRef> и ссылкой, полученной в результате процесса обнаружения, можно просмотреть WSDL-документ в браузере, как показано на рисунке справа.



Нет никакой необходимости работать с этой информацией непосредственно. WSDL-документ будет сгенерирован динамически с атрибутом `WebMethod`; к рассмотрению названного атрибута мы приступим ниже в этой главе. Добавление Web Reference с помощью Visual Studio.NET на стороне клиента позволяет не только получить информацию для обнаружения web-служб, которые поддерживаются данным сервером, но и запросить WSDL-документ. Этот WSDL-документ, в свою очередь, используется для создания клиентского посредника, обладающего теми же самыми методами и аргументами. Использование такого посредника клиентским приложением обладает тем преимуществом, что теперь у приложения есть возможность вызывать все методы так, как если бы они были реализованы на сервере, поскольку посредник принимает на себя выполнение преобразования этих вызовов в SOAP-вызовы и обращение к методам по сети.

Спецификации поддерживаются консорциумом World Wide Web Consortium (W3C). С этими спецификациями можно ознакомиться на web-сайте W3C: www.w3.org/TR/wsdl.

Вызов метода



Для вызова метода web-службы необходимо преобразовать обращение к методу в SOAP-сообщение в соответствии с описанием, содержащимся в WSDL-документе. На рисунке слева представлены составные части SOAP-сообщения. Это сообщение состоит из SOAP-конверта, который объединяет всю SOAP-информацию в общий блок. Сам конверт состоит из двух частей — SOAP-заголовка и SOAP-тела. Заголовок является необязательным и позволяет описывать, каким образом клиент и сервер должны обрабатывать тело сообщения. Тело является обязательным и состоит из блоков вызываемых методов. В SOAP-теле клиент кроме собственно вызова метода посылает сериализованные значения всех необходимых методу аргументов. SOAP-сервер отправляет обратно возвращаемые значения в теле ответного SOAP-сообщения (см. рис. слева).

В следующем примере демонстрируется, каким образом выглядит SOAP-сообщение, которое посылается от клиента серверу. Клиент вызывает метод web-службы `ReverseString()`. В качестве аргумента этому методу передается строка `Hello World!`. Заметьте, что обращение к методу находится в теле SOAP-сообщения, внутри XML-элемента `<soap:Body>`. Само тело находится внутри конверта: `<soap:Envelope>`. Перед самим SOAP-сообщением присутствует HTTP-заголовок, поскольку SOAP-сообщение отправляется в виде запроса HTTP POST.

Нет необходимости создавать это сообщение, поскольку оно автоматически генерируется клиентским посредником:

```
POST /WebServiceSample/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 508
SOAPAction: "http://www.wrox.com/webservice/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/webservice">
        <message>Hello World!</message>
      </ReverseString>
    </soap:Body>
  </soap:Envelope>
```

В ответ на обращение сервер отправляет назад SOAP-сообщение с результатом !dlroW olleH, который можно обнаружить в XML-элементе ReverseStringResult:

```
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/webservice">
      <ReverseStringResult>!dlroW olleH</ReverseStringResult>
    </ReverseStringResponse>
  </soap:Body>
</soap:Envelope>
```

Спецификации SOAP поддерживаются Рабочей группой по протоколу XML компании W3C (при необходимости обращайтесь на сайт www.w3.org/TR/soap и www.w3.org/TR//2001/WD-soap12-20010709 за версией 1.2).

SOAP и межсетевые экраны

Системные администраторы часто задают вопрос, позволяет ли протокол SOAP преодолевать границы обеспечения безопасности межсетевой защиты или, другими словами, нарушает ли SOAP концепцию межсетевых экранов? На самом деле, в данном случае возникают те же самые проблемы, связанные с обеспечением безопасности, что и для обычных web-серверов. При работе с обычным сервером администратор межсетевого экрана открывает порт HTTP 80, позволяя серверу взаимодействовать через него с внешним миром. Пользователи, находящиеся в Интернете, получают возможность прямого доступа к этим серверам, несмотря на то, что они находятся за стеной межсетевого экрана. Пользователь может отправить серверу HTML-файл с HTTP-запросом, а сервер — в качестве ответа — может отправить либо статическую страницу, либо страницу, которая была создана динамически посредством ASP- или CGI-сценариев. Web-службы — это всего лишь еще один тип приложения, выполняющегося на стороне сервера, которое для взаимодействия использует HTTP, хотя вместо того, чтобы отправлять и принимать простые запросы HTTP GET и HTTP POST, оно принимает запрос HTTP POST, где содержится встроенное SOAP-сообщение, а вместо того, чтобы возвращать HTML, оно возвращает HTTP-ответ с встроенным ответным SOAP-сообщением. Что касается межсетевого экрана, то все взаимодействие осуществляется посредством HTTP, следовательно, оно может проходить через порт 80.

Если web-служба работает не совсем так, как требуется, например, не обеспечивает сохранность конфиденциальной информации или нарушает нормальную работу сервера, то это приведет к возникновению определенных проблем, однако такие проблемы являются совершенно обычными для любых приложений, выполняющихся на стороне сервера, будь то традиционные web-страницы, серверные коммерческие объекты или web-службы.

Если системный администратор межсетевого экрана все равно опасается ненадлежащей организации безопасности web-служб, то он может воспользоваться фильтром приложений, чтобы запретить SOAP-вызовы в HTTP-запросах.

Web-службы и .NET Framework

.NET Framework существенно облегчает создание и использование web-служб. Ниже перечислены четыре основные пространства имен, предназначенные для работы с web-службами:

- Классы, описанные в пространстве имен `System.Web.Services`, используются при создании web-служб.
- При помощи пространства имен `System.Web.Services.Description` есть возможность описывать web-службы посредством WSDL.
- При обнаружении web-служб с использованием протокола DISCO применяются классы из пространства имен `System.Web.Services.Discovery`.
- С помощью пространства имен `System.Web.Services.Protocols` можно создавать SOAP-запросы и ответы.

Ниже мы рассмотрим использование всего лишь двух пространств имен — `System.Web.Services.Description` и `System.Web.Services.Discovery`. Остальные два пространства имен мы только затронем в этой главе. Более подробная информация может быть найдена в книге "ASP.NET Programmer's Reference" *(Руководство программиста по ASP.NET), ISBN 186100530X, выпущенной издательством WROX.*

Создание web-службы

Для реализации web-службы можно создать класс web-службы, который является производным от пространства имен `System.Web.Services.WebService`. Класс `WebService` обеспечивает доступ к объектам ASP.NET `Application` и `Session`:

Свойство WebService	Описание
`Application`	Получает объект `HttpApplicationState` для текущего запроса.
`Context`	Получает объект `HttpContext`, в котором заключена информация, относящаяся к HTTP. С помощью этого объекта может быть прочитана информация в HTTP-заголовке.
`Server`	Получает объект `HttpServerUtility`. Этот класс обладает несколькими вспомогательными методами, которые позволяют зашифровывать и расшифровывать URL.
`Session`	Получает объект `HttpSessionState`, который используется для сохранения некоторого состояния клиента.
`User`	Получает пользовательский объект, реализующий интерфейс `IPrincipal`. С помощью данного интерфейса есть возможность узнать имя пользователя и тип аутентификации.

Атрибут WebService

Подкласс класса `WebService` должен быть помечен атрибутом `WebService`. Класс `WebServiceAttribute` обладает следующими свойствами:

Свойство	Описание
`Description`	Описание web-службы, которое будет использовано в WSDL-документе.
`Name`	Позволяет считывать или задавать название web-службы.
`Namespace`	Позволяет считывать или задавать пространство имен XML для данной web-службы. По умолчанию используется значение `http://tempuri.org`, которое вполне подходит для нужд тестирования, однако перед тем как сделать службу общедоступной, это пространство имен должны быть изменено.

Мы познакомились с атрибутами в главе 22. Здесь снова можно увидеть атрибуты в действии.

Атрибут WebMethod

Все методы, предоставляемые web-службой, должны быть помечены атрибутом `WebMethod`. Web-служба может обладать и методами, которые не помечаются атрибутом `WebMethod`. Такие методы могут вызываться из web-методов, но при этом они не могут быть вызваны клиентом. Вызывать метод удаленным клиентам позволяет использование атрибута класса `WebMethodAttribute`, при этом можно определять, должна ли осуществляться буферизация ответов, какова допустимая длина кэша и должно ли сохраняться состояние сеанса в виде именованных параметров. В следующей таблице перечислены свойства класса `WebMethodAttribute`.

Свойство	Описание
`BufferResponse`	Позволяет устанавливать или сбрасывать флажок, указывающий на необходимость буферизации ответа. Значение по умолчанию — `true`. При сохранении ответа в буфере пользователю будет отправляться только полностью завершенный пакет.
`CacheDuration`	Позволяет устанавливаться интервал времени, на протяжении которого результат будет храниться в кэше. Если тот же самый запрос будет повторен в течение определяемого данным свойством промежутка времени, то будет возвращаться ответ, хранящийся в кэше. По умолчанию значение данного свойства равно 0, что означает, что результат вообще не кэшируется.
`Description`	Содержит описание, которое используется для создания help-страниц web-службы для будущих потребителей.
`EnableSession`	Логическое значение, определяющее, является ли доступным состояние сеанса. По умолчанию равно `false`; это означает, что свойство `Session` класса `WebService` не может быть использовано для сохранения состояния сеанса.
`MessageName`	По умолчанию имени сообщения присваивается название метода.
`TransacionOption`	Определяет, каким образом данный метод работает с транзакциями. По умолчанию используется значение `Disabled`.

Клиент

Чтобы вызвать метод, клиенту необходимо создать TCP-соединение с сервером web-службы и отправить HTTP-запрос для передачи SOAP-сообщения. При этом сам вызов метода должен быть преобразован в SOAP-сообщение. Все эти действия выполняет посредник на стороне клиента. Посредник на стороне клиента реализован в классе `SoapHttpClientProtocol`.

Класс SoapHttpClientProtocol

Класс `System.Web.Services.Protocols.SoapHttpClientProtocol` является базовым классом для клиентского посредника. Метод `Invoke()` позволяет привести аргументы к виду, необходимому для создания SOAP-сообщения, которое будет отправлено web-службе. То, какая именно служба будет вызвана, определяется свойством `Url`.

Класс `SoapHttpClientProtocol` также поддерживает вызов web-служб в асинхронном режиме с помощью методов `BeginInvoke()` и `EndInvoke()`.

Альтернативные клиентские протоколы

Помимо класса `SoapHttpClientProtocol` могут также использоваться и другие классы-посредники. Протоколы `HttpGetClientProtocol` и `HttpPostClientProtocol` позволяют отправлять простые запросы HTTP GET и HTTP POST, не несущие дополнительной нагрузки в виде SOAP-вызова.

> *Классы `HttpGetClientProtocol` и `HttpPostClientProtocol` могут применяться только тогда, когда в вашем решении используется .NET как со стороны клиента, так и со стороны сервера. Если вы хотите поддерживать различные технологии, то вам следует использовать протокол SOAP.*

Сравните запрос HTTP POST, приведенный ниже, с SOAP-вызовом, встретившимся в этой главе ранее:

```
POST /WebServiceSample/Service1.asmx/ReverseString HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

message=string
```

Запрос HTTP GET оказывается еще короче. Недостатком запроса GET является ограничение, налагаемое на размер передаваемых параметров. Если размер выходит за пределы 1 Кбайт, то следует воспользоваться запросом POST:

```
POST /WebServiceSample/Service1.asmx/ReverseString?message=string HTTP/1.1
Host: localhost
```

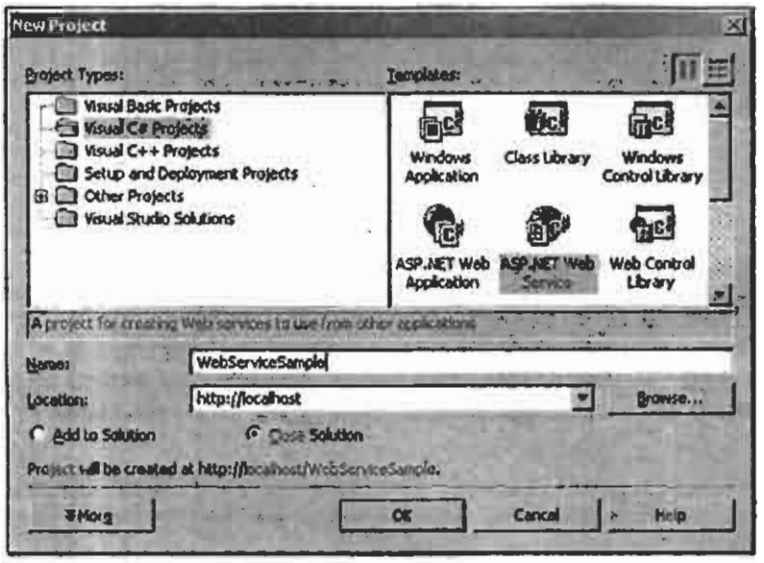
При использовании протоколов `HttpGetClientProtocol` и `HttpPostClientProtocol` накладные расходы по сравнению с использованием SOAP-методов будут меньше; недостатком этих протоколов является тот факт, что они не поддерживаются web-службами, работающими на других платформах.

Создание простой web-службы ASP.NET

Создадим простую web-службу с помощью Visual Studio.NET.

Практикум: проект по созданию web-службы

1. Создайте новый проект с помощью пункта меню File | New | Project и выберите шаблон ASP.NET Web Service, как показано на рисунке на стр. 739. Присвойте проекту имя `WebServiceSample` и нажмите кнопку OK.



Автоматически создаваемые файлы

Ниже перечислены файлы, создаваемые программой-мастером:

- Как нам известно по предшествующим главам, файл `AssemblyInfo.cs` используется для хранения атрибутов модуля.
- `Global.asax`: запросы на запуск и на окончание приложения сеанса — точно такие же, как и для приложения Web Forms, которое было создано в предыдущей главе.
- В файле `Service1.asmx` хранится класс нашей web-службы. Все web-службы ASP.NET могут быть идентифицированы по расширению `.asmx`. Файл, в котором содержится исходный код, носит название `Service1.asmx.cs`, поскольку в Visual Studio.NET используется код поддержки. Однако этот файл по умолчанию в Solution Explorer не является видимым, и, для того чтобы до него добраться, необходимо щелкнуть мышью на кнопке **Show All Files** (вывод всех файлов), расположенной на панели инструментов Solution Explorer.

  Мастер генерирует класс `Service1`, который является производным от класса `System.Web.Services.WebService`. В файле `Service1.asmx.cs` можно также увидеть некоторый образец кода, позволяющий представить себе, как должен кодироваться метод web-службы — он должен быть описан как общий и быть помечен атрибутом `WebMethod`:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WebServiceSample
{
    /// <summary>
    /// Краткое описание класса Service1.
    /// </summary>
    public class Service1 : System.Web.Services.WebService
    {
        // ПРИМЕР WEB-СЛУЖБЫ
        // Используемая в качестве примера служба HelloWorld() возвращает строку
        // Hello World.
        // Для ее создания удалите знаки комментария из последующих строк,
        // сохраните файл и постройте проект
        // Протестировать эту web-службу можно, нажав F5
```

```
//    [WebMethod]
//    public string HelloWorld()
//    {
//        return "Hello World";
//    }
}
```

- `Web.config` — это файл конфигурации для данного приложения. Как говорилось в главе 24, можно изменять конфигурацию данного файла, настраивая трассировку, состояние сеанса и режим отладки.
- `WebServiceSample.vsdisco` — это файл обнаружения для данного приложения. XML-элемент `<dynamicDiscovery>` означает, что поиск web-служб будет осуществляться во всех директориях за исключением указанных в элементе `<exclude>`:

```
<?xml version="1.0" ?>
<dynamicDiscovery xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">
<exclude path="_vti_cnf" />
<exclude path="_vti_pvt" />
<exclude path="_vti_log" />
<exclude path="_vti_script" />
<exclude path="_vti_txt" />
<exclude path="Web References" />
</dynamicDiscovery>
```

Добавление web-метода

Следующее, что необходимо сделать, это добавить некоторый метод в web-службу. Мы добавим в нее простой метод `ReverseString()`, который получает строку и возвращает клиенту перевернутую строку.

Практикум: добавление метода

1. Добавьте следующий код в файл `Service1.asmx.cs`:

```
[WebMethod]
public string ReverseString(string message)
{
    char[] arr = message.ToCharArray();
    Array.Reverse(arr);
    message = new string(arr);

    return message;
}
```

Чтобы однозначно идентифицировать XML-элементы в создаваемом описании web-службы, необходимо добавить пространство имен. Добавим атрибут `WebService` с пространство имен **http://www.wrox.com/webservices** в класс **Service1**. Можно использовать любую другую строку, которая идентифицирует XML-элементы уникальным образом. Так, например, можно использовать для этой цели URL-ссылку на страницу вашей компании. Причем такая web-ссылка совершенно не обязательно должна существовать на самом деле; она просто используется для уникальной идентификации. Если вами используется пространство имен, которое основывается на web-адресе вашей компании, то почти наверняка можно гарантировать, что никакая другая компания не будет использовать то же пространство имен.

Если вы не задаете никакого пространства имен, то по умолчанию используется пространство имен **http://tempuri.org**. Для целей обучения это пространство нас

вполне устроит, однако никогда не следует устанавливать реально используемую web-службу с таким пространством имен.

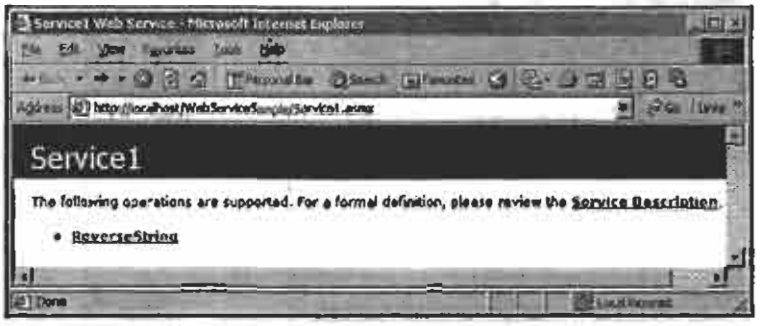
2. В свете вышесказанного измените пример следующим образом:

```
[WebService(Namespace="http://www.wrox.com/webservices")]
public class Service1 : System.Web.Services.WebService
{
```

3. Теперь можно откомпилировать проект.

Тестирование web-службы

Открыв файл `Service1.asmx` в браузере (его можно запустить из Visual Studio.NET с помощью пункта меню Debug | Start Without Debugging), вы получите перечень всех методов выбранной web-службы, как показано на рисунке справа. В данном случае, служба обладает только одним методом — `ReverseString()`.



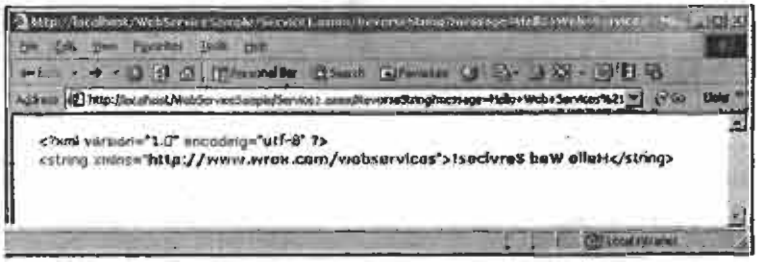


Выбор показаной на экране ссылки на метод `ReverseString` повлечет за собой переход в диалог, позволяющий проводить тестирование web-службы. Диалог обладает полями, где можно редактировать все параметры, передаваемые данному методу, — в рассматриваемом случае, это единственный параметр.

На этой странице мы также получаем информацию о том, как выглядят SOAP-вызовы, поступающие от клиента, и ответы, отправляемые сервером. На рисунке слева приводится пример использования SOAP, HTTP GET и HTTP POST.

Нажав кнопку `Invoke`, после того как в текстовое окно будет введен текст `Hello web services!`, мы получим от сервера результат, представленный на рисунке справа.

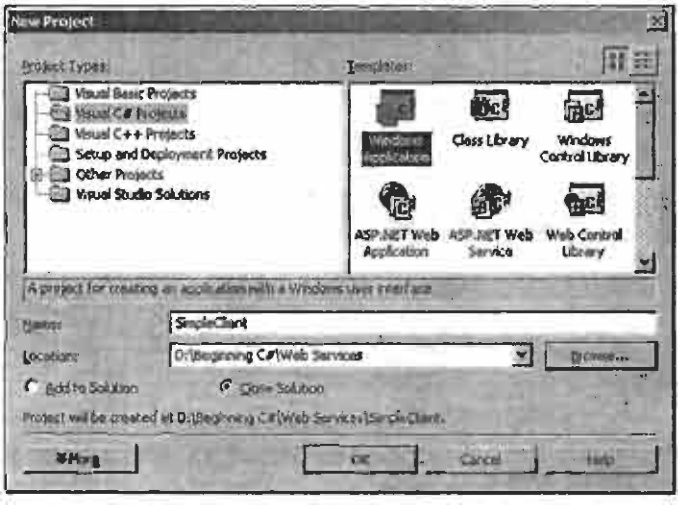
Результат имеет тип `string`, и, как и предполагалось, это — инверсирование введенной строки.



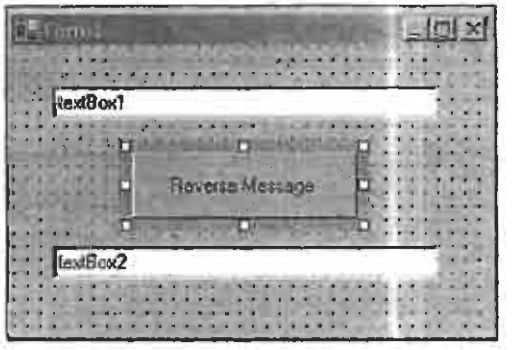
Реализация программы-клиента

Тестовый вариант работает, и теперь создадим Windows-клиента, который сможет воспользоваться этой web-службой. Клиент должен создать SOAP-сообщение, которое будет отправлено по HTTP-каналу. Это совсем не обязательно выполнять нам самим. Класс `System.Web.Services.Protocols.SoapHttpClientProtocol` выполняет всю эту работу скрытно от нас.

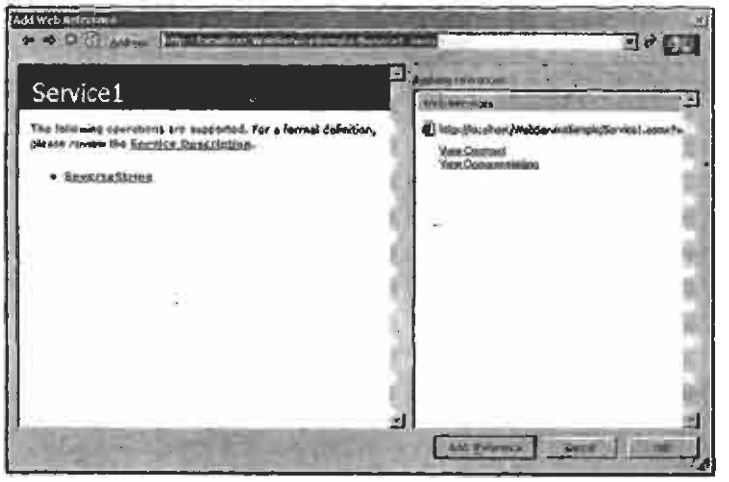
Практикум: создание клиентского приложения Windows

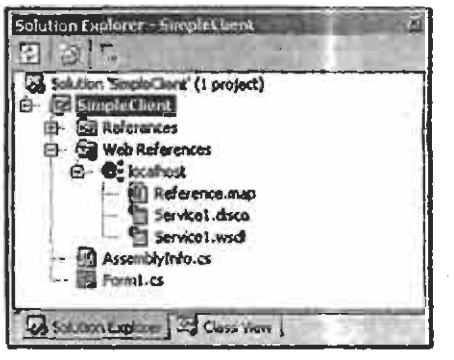


1. Создайте новый проект C# — приложение Windows (см. рис. слева).
2. Добавьте в форму два текстовых окна и кнопку, как показано на рисунке ниже. Обработчик события "щелчок на кнопке" будет вызывать web-службу.



3. Добавьте web-ссылку, воспользовавшись пунктом меню Project | Add Web Reference, и выберите гиперссылку Web References on Local Service, чтобы перейти к предварительно созданной web-службе. Теперь, до того как вы нажмете кнопку Add Reference (см. рис. слева), вы можете просматривать контракт и документацию. Вместо того, чтобы использовать браузер, вы также можете непосредственно ввести URL `.asmx`-файла.

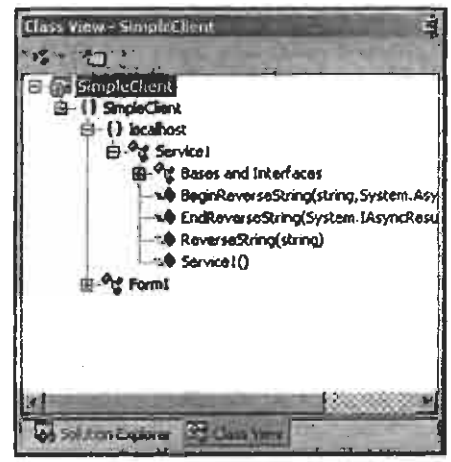




Как это работает

Теперь можно увидеть новую web-ссылку на localhost, а также соответствующие документы DISCO и WSDL, которые были отправлены клиенту (см. рис. справа).

В Solution Explorer нельзя увидеть, что был создан новый класс, в котором реализован посредник на стороне клиента, преобразующий вызовы методов в формат SOAP. Чтобы просмотреть этот класс, необходимо переключиться в окно Class View. Там можно увидеть, что создано новое пространство имен, включающее в себя имя сервера, в данном случае — localhost. Класс `Service1` является производным от класса `System.Web.Services.Protocols.SoapHttpClientProtocol` и в нем реализуется метод `ReverseString()`, предоставляемый web-службой.



Щелкните два раза мышью на методе `ReverseString()`, для того чтобы открыть автоматически сгенерированный файл `service1.cs`. Рассмотрим код, который был создан программой-мастером.

```
//------------------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.       /Этот код сгенерирован инструментом./
//     Runtime Version: 1.0.2914.16
//
//     Changes to this file may cause incorrect behavior and will be lost if
                              /Любые изменения, внесенные в этот файл, могут привести
                               к неправильной работе программы/
//     the code is regenerated.
                              / и будут потеряны, если код будет сгенерирован повторно./
// </autogenerated>
//------------------------------------------------------------------------------

namespace SimpleClient.localhost {
    using System.Diagnostics;
    using System.Xml.Serialization;
    using System;
    using System.Web.Services.Protocols;
    using System.Web.Services;

    [System.Web.Services.WebServiceBindingAttribute(Name="Service1Soap",
       Namespace="http://www.wrox.com/webservices")]
    public class Service1 :
       System.Web.Services.Protocols.SoapHttpClientProtocol {
```

В конструкторе свойству `Url` присваивается значение web-службы. Это свойство будет использоваться в классе `SoapHttpClientProtocol` для обращения к web-службе:

```
        [System.Diagnostics.DebuggerStepThroughAttribute()]
        public Service1() {
            this.Url = "http://localhost/WebServiceSample/Service1.asmx";
        }
```

Наиболее важным является метод, который собственно и предоставляет web-служба,— `ReverseString()`. В данном случае метод имеет тот же самый параметр, что и в реализации этого метода на сервере. Реализация метода `ReverseString()` на стороне клиента обращается к методу `Invoke()` базового класса `SoapHttpClientProtocol`. Этот метод создает SOAP-сообщение, в котором используется имя `ReverseString` и параметр `message`:

```
        [System.Diagnostics.DebuggerStepThroughAttribute()]
        [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
            "http://www.wrox.com/webservices/ReverseString",
            RequestNamespace="http://www.wrox.com/webservices",
            ResponseNamespace="http://www.wrox.com/webservices",
```

```
        Use=System.Web.Services.Description.SoapBindingUse.Literal,
        ParameterStyle=
            System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
    public string ReverseString(string message)
    {
        object[] results = this.Invoke("ReverseString", new object[]
                                        (message));
        return ((string)(results[0]));
    }
```

Клиент также может вызывать web-службу асинхронно. У метода `BeginReverseString()` имеются только те параметры, которые отправляются на сервер, а метод `EndReverseString()` возвращает полученный результат:

```
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    public System.IAsyncResult BeginReverseString(string message,
                        System.AsyncCallback callback, object asyncState)
    {
        return this.BeginInvoke("ReverseString", new object[] {
                                    message}, callback, asyncState);
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    public string EndReverseString(System.IAsyncResult asyncResult)
    {
        object[] results = this.EndInvoke(asyncResult);
        return ((string)(results[0]));
    }
}
}
```

До этого момента нам не пришлось написать самостоятельно ни единой строки кода для клиента. Мы создали простой пользовательский интерфейс и использовали меню Add Web Reference для создания класса посредника. Теперь надо связать их между собой.

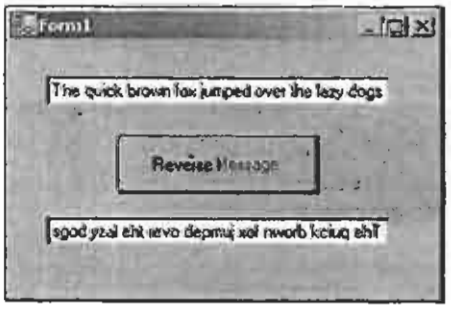
4. Добавьте обработчик событий `Click` для кнопки и включите следующие две строки кода:

```
private void button1_Click(object sender, System.EventArgs e)
{
    localhost.Service1 ws = new localhost.Service1();
    textBox2.Text = ws.ReverseString(textBox1.Text);
}
```

Как это работает

В следующей строке мы создаем новый экземпляр класса посредника. Как видно в реализации конструктора, свойству `Url` присваивается web-служба:

```
localhost.Service1 ws = new localhost.Service1();
```



При вызове метода `ReverseString()` на сервер отправляется SOAP-сообщение, что приводит к вызову web-службы:

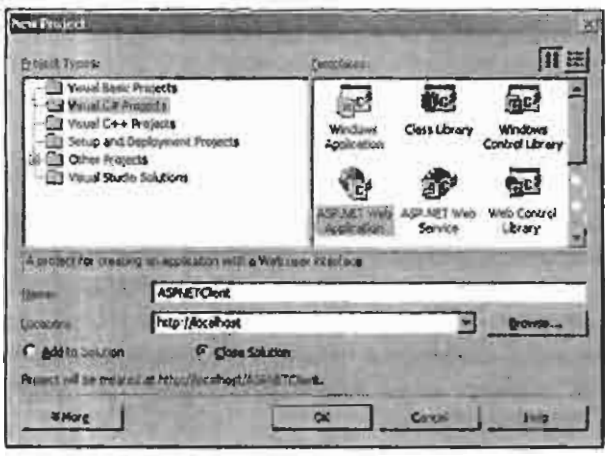
```
textBox2.Text = ws.ReverseString(textBox1.Text);
```

Запустив программу, мы получим результат, проиллюстрированный на рисунке слева.

Реализация клиента ASP.NET

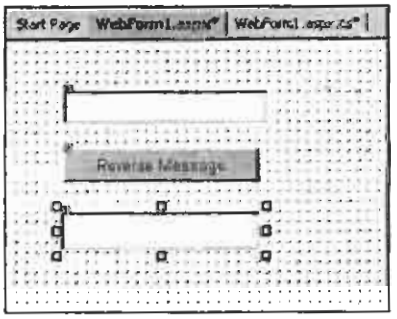
Та же web-служба может быть использована и из клиентского приложения ASP.NET. Как показано далее, задать ссылку на web-службу можно точно таким же образом, как и для приложения Windows.

Практикум: создание клиентского приложения ASP.NET



1. Создайте новый проект C# — ASP.NET Web Application (см. рис. слева). Назовите проект **ASPNETClient**.

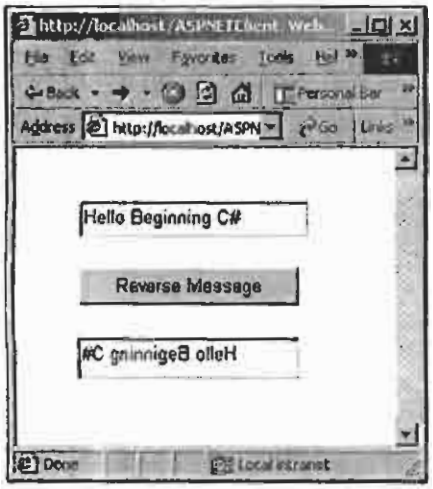
   Добавьте в форму два текстовых окна и одну кнопку, как показано на приведенном ниже рисунке. Мы будем вызывать web-службу с помощью обработчика событий `Click` этой кнопки.



2. Точно так же, как мы делали это для приложения Windows, добавьте web-ссылку **http://localhost/webservicesample/service1.asmx**.

3. После добавления web-ссылки снова будет создан класс посредника. Добавьте обработчик события "щелчок мышью на кнопке" и введите в него следующую программу:

```
private void Button1_Click(object sender, System.EventArgs e)
{
    ASPNETClient.localhost.Service1 ws =
                                new ASPNETClient.localhost.Service1();
    TextBox2.Text = ws.ReverseString(TextBox1.Text);
}
```



4. Постройте проект.

5. С помощью пункта меню **Debug | Start** вы можете запустить браузер и ввести тестовое сообщение в первое текстовое окно. Нажав кнопку, вы вызовете web-службу и получите во втором текстовом окне перевернутую строку, как показано на рисунке слева.

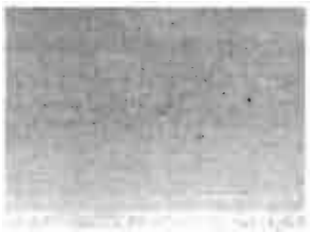
Использование web-служб в web-приложениях оказывается таким же простым, как и использование web-служб в приложениях Windows!

Итоги

В этой главе мы познакомились с тем, что представляют собой web-службы, и коротко рассмотрели протоколы, которые используются для работы с ними. Для того чтобы определить местоположение web-службы и запустить ее, надо выполнить следующее:

- **Директория** — найти web-службу посредством UDDI
- **Обнаружение** — выявить web-службы на хорошо известном сервере
- **Описание** — WSDL описывает методы и аргументы web-службы
- **Вызов** — осуществляются независимые от платформы вызовы методов с использованием протокола SOAP

Мы познакомились с тем, насколько простым оказывается создание web-служб в Visual Studio.NET, где класс `WebService` используется для определения соответствующих методов с помощью атрибута `WebMethod`. Создать клиента, который пользуется услугами web-служб, оказывается таким же простым делом, как и создание самих web-служб: требуется добавить в клиентский проект web-ссылку и использовать посредника. Ядром клиента является класс `SoapHttpClientProtocol`, который преобразует вызов метода в SOAP-сообщение.



Практический пример 1 — web-сайт для проведения опросов

На протяжении книги вы знакомились со всем спектром технологий .NET, которые можно использовать, программируя на C#. В этом разделе мы собираемся отойти от теории и заняться в качестве практического упражнения приложением, в котором будут использоваться самые разнообразные технологии из имеющихся в .NET.

Сначала мы создадим базовое приложение, а затем расширим его и создадим вторую версию, в которую будут включены дополнительные возможности. В реальной жизни от любого приложения всегда требуется постоянно соответствовать изменяющимся требованиям, и совершенно очевидно, что познакомиться с процессом изменения приложения в практическом примере такой книги, как эта, оказывается очень полезным. Использование двух различных версий также предоставит возможность узнать, каким образом .NET позволяет упростить процедуру расширения приложений, это означает что можно акцентировать внимание на меньших частях приложения в процессе его разработки и создания.

Приложение, в первую очередь, предназначается для работы с web-сайтом. Однако, как вы сможете увидеть по мере изучения этого примера, оно будет работать с любым типом интерфейсов. Важно помнить, что, хотя мы будем заниматься созданием приложения как функциональной возможности web-сайта, лежашая в его основе объектная модель будет работать не менее эффективно и для консольного приложения, и для desktop-приложения, и даже для web-службы. Мы также узнаем, каким образом трехзвенная модель приложения (в которой уровни данных, бизнес-логики и представления отделены друг от друга) позволяет упрощать процесс модернизации программного обеспечения на всем протяжении его жизненного цикла.

Web-сайт для проведения опросов

Многие web-сайты предусматривают вывод на экран вопроса с возможными вариантами ответов. Посетителям web-сайтов таким образом дается подсказка для ответа в соответствии с их точкой зрения, после чего они получают возможность просмотреть общие результаты текущего опроса. Рассмотрим пример:

```
Какой язык программирования вы, в основном, используете?

C#
C++
Java
JavaScript
Visual Basic
```

Результат, который увидит посетитель, может быть представлен примерно следующим образом:

```
10%   C#
 2%   C++
34%   Java
12%   JavaScript
42%   Visual Basic
```

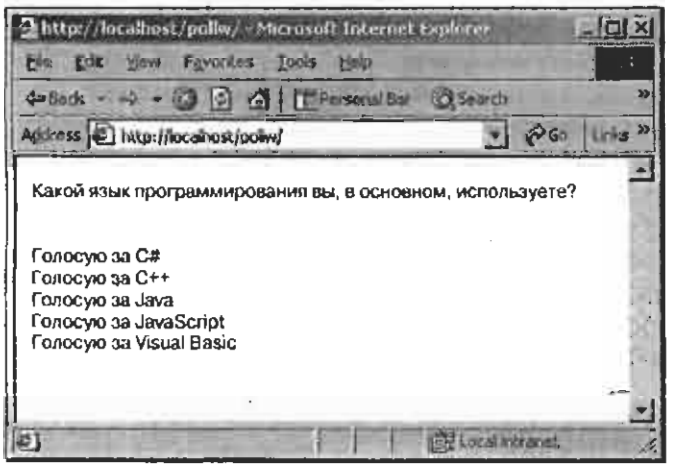
Практическим примером, которому посвящен этот раздел, будет реализация системы проведения опросов, подобных приведенному выше, и после того, как мы создадим две версии системы, мы включим в них новые возможности. Основным будет интерфейс, использующий web и построенный с помощью ASP.NET. Мы также создадим уровни представления для консольного приложения, desktop-приложения Windows и web-службы.

Версия 1

В первой версии web-сайта для проведения опросов будет реализован простой вывод вопроса на web-страницу и сбор ответов. Прежде чем двигаться дальше, ясно и четко сформулируем требования к этому web-сайту.

Требования

Наш не очень интенсивно посещаемый web-сайт должен включать в себя опрос, который состоит из вопроса и нескольких возможных вариантов ответа (обычно не более десяти). При проведении опроса посетитель выбирает один из предложенных ответов. После того как посетитель проголосовал, он получает возможность увидеть число голосов, отданных за каждый из вариантов.



Все детали, касающиеся вопроса, ответов и подсчета голосов, хранятся в XML-файле на web-сервере, поскольку бюджет на БД не выделен, а возможность должна быть реализована в срочном порядке. Когда администратору требуется внести какие-либо изменения в проведение опроса, он вручную редактирует XML-файл. Приложение должно выводить простой HTML так, чтобы он был корректно представлен в двух основных типах браузеров (Netscape и Internet Explorer).

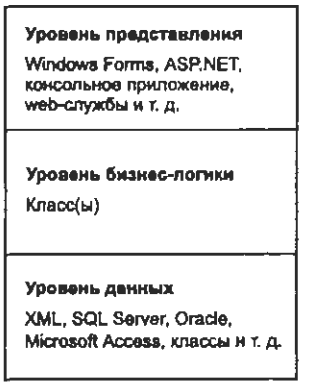
Разработка объектной модели (Business Logig Tier)

Для данного примера будет использоваться трехзвенная модель — мы собираемся провести четкую границу между всеми тремя уровнями приложения:

- **Уровень представления.** Этот уровень представляет собой пользовательский интерфейс и для .NET может состоять из приложения ASP.NET (Интернет, интранет, цифровое телевидение, мобильный телефон, электронная записная книжка и т. д.), консольного приложения, приложения Windows Forms или web-службы.

В данном примере мы планируем рассмотреть создание уровня представления с использованием приложения ASP.NET, приложения Windows Forms, web-службы и консольного приложения с командной строки.

- **Уровень бизнес-логики.** Это сердцевина приложения. Именно здесь мы будем применять принципы объектно-ориентированной разработки в чистом виде, и именно здесь будет располагаться бóльшая часть создаваемого кода.
- **Уровень данных.** Это уровень, на котором будут храниться данные, будь это XML-файл, обычный файл на диске либо БД SQL Server, Microsoft Access, MSDE или Oracle, либо какое-либо другое средство хранения данных. Во многих приложениях — кроме собственно среды хранения данных — на этом же уровне располагаются и классы, которые позволяют осуществлять доступ к данным (см. рис. справа).



Необходимо помнить о том, что в .NET имеется целый спектр мощных и гибких инструментов, однако для создания высококачественного продукта прежде всего играют роль методы, которые используются при разработке приложений, а не инструменты сами по себе. Вполне возможно создать такой сайт для проведения опросов, используя только `.aspx`-файлы, которые непосредственно привязаны к хранилищу данных. Однако в данном примере используется более структурированный подход, который приводит к созданию приложения, разбитого на три логические составные части; такое приложение гораздо проще совершенствовать и расширять. Время, затрачиваемое на создание программного обеспечения, намного меньше времени на сопровождение этого программного обеспечения; именно поэтому структурированный подход представляет особую ценность.

Для трехзвенной структуры приложения представляется благоразумным учитывать предъявляемые к нему требования с самого начала разработки объектной модели. Это означает сначала построение связной умозрительной модели того, из каких "вещей" должно состоять приложение, а затем создание на основании этой модели уровня бизнес-логики приложения. Именно на этом этапе создаются объекты, моделирующие отдельные составные части проблемы из той области знаний, в которой мы работаем, и мы строим эти объекты так, чтобы соответствовать требованиям уровня представления и разумно использовать уровень данных. Зачастую возникают ситуации, когда по мере разработки объектов, находящихся в среднем звене, нам приходится принимать во внимание особенности физической среды выполнения (например, быстродействие), однако в данном конкретном случае мы разрабатываем очень простую систему, поэтому физические особенности среды не играют существенной роли, и можно сосредоточить основное внимание на разработке объектов уровня бизнес-логики. Это дает возможность создать умозрительную модель в чистом виде и впоследствии расширять ее по мере изменения требований.

Возможные классы

При создании объектной модели зачастую используется смесь интуиции и практического опыта, хотя для этого обычно существует теоретическое обоснование, которое вырабатывается с течением времени. Начнем с того, что еще раз просмотрим

все требования и перечислим все имена существительные, которые мы обнаружим в описании классов/объектов нашего приложения.

- Web-сайт
- Опрос
- Вопрос
- Ответ
- Посетитель сайта
- Голосование

В этом списке наиболее заслуживающими внимания представляются следующие пункты: опрос, вопрос, ответ, голосование. Оставшиеся два (web-сайт и посетитель сайта) не являются специфичными именно для сайта, предназначенного для сбора голосов, хотя они могут понадобиться позднее.

Класс Poll (опрос)

Когда мы обсуждаем понятие опроса, мы говорим о весьма специфическом наборе данных:

- Единственный вопрос
- Множество ответов
- Подсчет голосов для каждого из возможных ответов

Не следует забывать, что в XML-файле может находиться несколько различных вариантов опроса, и нужно быть в состоянии создавать объект `Poll` для каждого из вариантов.

Когда мы представляем себе, какие данные придется хранить в объектах `Poll`, рассмотрим какие функциональные возможности потребуются от такого объекта:

- **Голосование** — кажется логичным предусмотреть метод `Vote()` (голосование), который будет регистрировать выбор одного из ответов при проведении опроса.
- **Получение вопроса** — необходимо предусмотреть возможность доступа к вопросу, чтобы вывести его на web-странице.
- **Получение ответа** — необходимо считывать список возможных при проведении опроса ответов, которые будут выводиться на странице для выбора посетителями сайта.
- **Получение числа голосов для данного варианта ответа** — необходимо предусмотреть способ выявления проголосовавших за данный вариант. Это будет использоваться для вывода количества голосов по каждому варианту после голосования очередного посетителя.

В классе `Poll` содержится бóльшая часть необходимых возможностей, однако следует рассмотреть и другие классы, чтобы выяснить, какие возможности они могут предложить.

Класс Question (вопрос)

Что представляет из себя вопрос? Насколько известно из наших требований, это просто строка типа "Какую платформу вы используете?". Поскольку это так просто, то можно не утруждать себя реализацией вопроса в качестве объекта,

а вместо этого представить его в качестве поля какого-либо другого класса. Окончательное решение мы примем, когда будем рассматривать отношения между классами.

Класс Answer (ответ)

Как и вопрос, ответ представляет из себя просто строку; однако не забывайте, что, хотя для каждого опроса существует только один вопрос, для него — и для каждого опроса— имеется несколько вариантов ответа. Следует также принять во внимание то, что каждый вариант ответа предполагает сохранение числа поданных за него голосов. Если бы мы пришли к выводу о необходимости реализации ответа в качестве класса, то нам следовало бы предусмотреть свойства для текста самого вопроса и для хранения числа поданных голосов, а также включить метод `Vote()`. Окончательно решение будет принято нами в следующем разделе.

Класс Vote (голосование)

Что такое голос? Это либо да, либо нет. Существует другой вариант, при котором можно говорить о голосовании как о классе "Votes"; в этом случае он мог бы содержать счетчики числа голосов для каждого из имеющихся ответов. При таком варианте сценария класс `Votes` представлял бы собой нечто большее, чем просто некий массив. В этом случае представляется бессмысленным создавать класс, который будет представлять из себя всего лишь простой массив.

Отношения между классами

Представляется совершенно очевидным, что в качестве центрального компонента приложения используется класс `Poll`. В нем будут содержаться собственно вопрос, набор возможных ответов и счетчики числа голосов. Кроме того, в нем предусмотрен метод `Vote()`, который будет регистрировать голоса, отдаваемые посетителями.

Едва ли имеет смысл создавать для вопроса отдельный класс, поскольку вопрос представляет собой простую строку, поэтому мы разместим его в поле, имеющем тип строки, внутри какого-либо другого класса. Поскольку у каждого объекта `Poll` имеется только один вопрос, то кажется логичным прийти к заключению, что поле, в котором содержится вопрос, должно находится внутри класса `Poll`.

Возникает вопрос, каким образом следует реализовать ответы и подсчет голосов, поскольку в данном случае есть выбор — создавать ли нам специальный класс "Answer" (ответ), в котором будут содержаться строка (с текстом ответа) и счетчик голосов, или просто включить соответствующую функциональную возможность в класс `Poll`, предусмотрев поле, где будет храниться семейство ответов, и параллельно еще одно поле, в котором будет храниться число голосов для каждого ответа (другими словами, второй элемент семейства поли, где находится счетчик голосов, будет содержать количество голосов, которое было подано за вариант ответа, описываемый вторым элементом в семействе ответов). В результате такого подхода с точки зрения объектной модели необходимо регистрировать голоса следующим образом:

```
myAnswerText = myPoll.Answrers[3];
myPoll.Vote(3);
```

В данном случае для регистрации голоса используется метод `Vote()` на объекте `Poll`. Также имеется свойство с именем `Answers` (ответы), которое обеспечивает доступ к семейству строк (возможно, хранящемуся в виде некоторого массива).

Если бы мы приняли решение создать класс Answer внутри класса Poll, то код, позволяющий выполнять те же самые действия, принял бы приблизительно следующий вид:

```
myAnswerText = myPoll.Answrers[3].Text;
myPoll.Answrers[3].Vote();
```

При желании можно записать его следующим образом:

```
myAnswer = myPoll.Answrers[3];
myAnswerText=myAnswer.Text;
myAnswer.Vote();
```

Отличие между двумя последними примерами заключается в том, что теперь семейство представляет собой семейство объектов Answer, каждый из которых обладает свойством Text (в котором содержится текст ответа) и методом Vote(). С практической точки зрения совершено очевидно, что первый вариант является более аккуратным; он позволит не тратить силы и время на создание класса Answer. Дополнительным преимуществом является то, что позднее, когда встанет задача расширения приложения, нам придется работать с меньшим количеством классов, и в такой редакции приложение, скорее всего, окажется более просто сопровождать. В силу этих причин мы реализуем ответы и счетчики голосов в виде семейств, являющихся полями класса Poll, а не в виде отдельного класса Answer. На самом деле, если впоследствии возникнет необходимость в таком классе, то нам ничего не сможет помешать создать его, но на данном этапе мы оставим все простым и понятным.

Объектная модель

Объектная модель приложения для проведения опросов пока состоит из единственного класса — Poll, который может быть представлен следующим образом:

- Метод — Vote()
- Свойство — Question
- Свойство — Answers[] (семейство)
- Свойство — Votes[] (семейство)

Примем, что ответы для данного опроса будут храниться в свойстве, возвращающем семейство строк, которое может просматриваться в цикле. Со счетчиками голосов работа будет вестись аналогичным образом, и они будут возвращать значения целого типа.

На данный момент имеется объект, который позволяет с успехом хранить сайт для проведения опросов, однако каким образом можно такие объекты создавать? Необходимо предусмотреть метод, позволяющий создавать объект Poll, который соответствовал бы активному в настоящий момент опросу. Воспользуемся для этой цели статическим методом, например, таким:

```
CurrentlyActivePoll = Poll.Current();
```

Это приведет нас к объектной модели, которая может быть представлена следующим образом:

- Метод — Vote()
- Метод — Current() (статический)

- Свойство — `Question`
- Свойство — `Answers[]` (семейство)
- Свойство — `Votes[]` (семейство)

Свойства и методы класса Poll

Рассмотрим сигнатуры каждого из методов и свойств более внимательно, чтобы четко представлять, как они должны быть реализованы. В следующем разделе мы рассмотрим, каким образом класс может быть использован в типичной ситуации.

Метод Vote()

Метод `Vote()` может получать целое значение в диапазоне от 1 до *n*, где *n* — это число ответов в данном опросе. Этот метод не возвращает значение:

```
public void Vote(int answer)
```

Статический метод Current()

Метод `Current()` является статическим, поэтому доступ к нему осуществляется в помощью ссылки на имя класса, а не на имя экземпляра данного класса. Параметры у него отсутствуют, и он возвращает значение типа `Poll`, которое представляет опрос, активный в настоящий момент:

```
public static Poll Current()
```

Свойство Question

Свойство `Question` возвращает строку, которая может быть использована для вывода вопроса посетителям web-сайта, например: "Какую платформу вы используете в работе?".

```
public string Question
```

В первой версии мы редактируем вопрос через XML-файл. Это означает, что данное свойство может использоваться в режиме "только чтение", поэтому необходимо реализовать предложение `get`, а не `set`.

Свойство Answers

Свойство `Answers` обеспечивает доступ к объекту, в котором содержится семейство ответов для опроса; при опросе по использованию операционных систем это могут быть ответы типа "Windows 2000", "Linux", "MacOS" и т. д. Поскольку нам известен размер семейства, а все входящие в него элементы имеют один и тот же тип, то представляется логичным воспользоваться типом стандартного массива и хранить ответы в виде массива строк:

```
public string [] Answers
```

Как и для свойства `Question`, все изменения в данные вносятся вручную посредством редактирования XML-файла, поэтому требуется реализовать только предложение `get` — это свойство также будет использоваться в режиме "только чтение".

Свойство Votes

Свойство `Votes` содержит в себе единственное целое значение для каждого из ответов проводимого опроса. Названное свойство возвращает массив целого типа, существующий параллельно с массивом `Answers`; между этими двумя массивами

устанавливается взаимно однозначное соответствие, которое должно поддерживаться нашей реализацией:

```
public int[] Votes
```

Значения, хранящиеся в свойстве `Votes`, могут изменяться только с помощью метода `Vote()`, поэтому данное свойство, так же как и свойства `Question` и `Answers`, будет использоваться в режиме "только чтение", и для него необходимо реализовать только предложение `get`.

В классах, подобных данному, использование одного массива должно быть синхронизировано с использованием другого. Возможно, в приложении, предполагающем коммерческое использование, будет более предпочтительным создать отдельный класс `Answer`, в котором текст ответа и счетчик голосов будут более тесно привязаны друг к другу; не исключено также, что вы сможете отыскать какое-либо иное решение. Чтобы не усложнять пример, мы продолжим использовать два параллельных массива, что в данном случае нас вполне устраивает, поскольку мы очень аккуратно будем относиться к тому коду, который производит изменения в семействах.

Типичное использование

Будем считать, что теперь есть объектная модель, но как ее можно использовать? Достаточно часто мы будем обнаруживать пробелы в нашей разработке при рассмотрении вопроса о том, каким образом используются методы и свойства, поскольку это может приводить к возникновению очевидных проблем, в частности, к концептуальной путанице относительно того, каким образом работает сама модель (возможно, вам приходилось слышать о таких проблемах, как варианты использования, которые являются важным элементом объектно-ориентированного программирования). Рассмотрим простое консольное приложение, которое берет текущий опрос, выводит его и регистрирует голосование. Это позволит понять, работает ли модель с концептуальной точки зрения:

```
Poll myPoll = Poll.Current ();
Console.WriteLine (myPoll.Question);
for(int i=1; i<myPoll.Answers.Length; i++)
{
   Console.Write (myPoll.Answers [i] + ": ");
   Console.WriteLine (myPoll.Votes [i] + " voted");
}
MyPoll.Vote (2);
Console.WriteLine ("Vote registered against " + myPoll.Answers [2]);
Console.WriteLine (Poll.Votes [2]);
```

Все выглядит просто замечательно; у нас есть простой и ясный путь, позволяющий считывать текущий опрос, выводить его и регистрировать процедуру голосования. Здесь используется `myPoll.Awnswers.Length` для определения количества ответов. Это выглядит довольно запутанным, поскольку совершенно неясно, используется ли в данном случае свойство `Length`, принадлежащее `Votes` или `Answers`, поэтому следует избавиться от этой путаницы и ввести свойство `AnswerCount`, которое будет возвращать число ответов для данного варианта опроса:

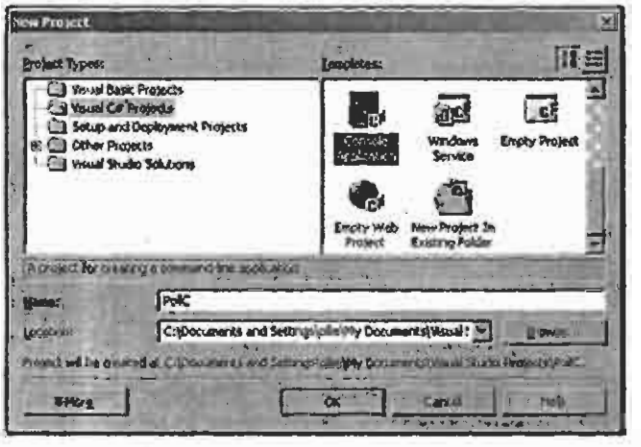
```
public int AnswerCount                    // Общее свойство класса Poll
```

Это приводит к некоторым изменениям в использовании класса `Poll` и позволяет сделать интерфейс более аккуратным:

```
Poll myPoll = Poll.Current ();
Console.WriteLine (myPoll.Question);
for (int i=1; i<myPoll.AnswersCount; ++i;
{
   Console.Write (myPoll.Answers [i] + ": ');
   Console.WriteLine (myPoll.Votes [i] + " votes");
}
MyPoll.Vote (2)
Console.WriteLine ("Vote registered against " = myPoll.Answer [2]);
ConsoleWriteLine (poll.Votes [2]);
```

Тестирование

Прежде чем двигаться дальше, создадим небольшое консольное приложение, которое будет тестировать класс `Poll`. У такого подхода есть два существенных преимущества. Во-первых, в процессе создания класса `Poll` всегда можно убедиться в том, что он работает. Во-вторых, можно понять, каким образом класс `Poll` допускает свое использование из консольного приложения с учетом того, что оно создается прежде всего для использования в приложении ASP.NET.



Сначала мы создаем новое консольное приложение C# в Visual Studio.NET с именем `PollC` (сокращение от Poll Command Line) (см. рис. слева).

Visual Studio.NET создаст проект консольного приложения и сгенерирует файл с именем `Class1.cs`, где находится статический метод `Main()`, который начнет выполняться в момент запуска файла данного консольного приложения с расширением `.exe`.

```
using System;

namespace PollC
{
   /// <summary>
   /// Общее описание для класса1
   /// </summary>
   class Class1
   {
      static void Main (string [] (args)
      {
         //
         // Действие: добавить код для запуска приложения
         //
      }
   }
}
```

Теперь добавим в консольное приложение требуемый код, который позволит считывать активный в настоящий момент опрос, выводить его вместе со счетчиками голосов, регистрировать процесс голосования и выводить всю информацию снова. Для этого потребуется внести в метод `Main()` следующие изменения:

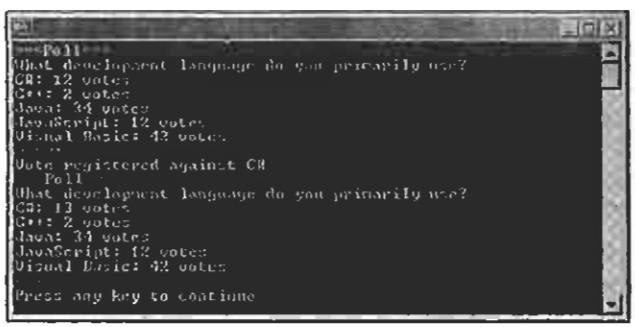
```
static void Main (string) [] args)
{
   Poll myPoll = POll.Current ();
```

```
        DisplayPoll (myPoll);
        myPoll.Vote (2);
        Console.WriteLine ("Vote registered against " + myPoll.Answers [2];
        DisplayPoll (myPoll);
    }
```

Приведенный код очень напомнинает тот код, который нам встретился в тот момент, когда мы рассматривали типичное использование класса `Poll`. В первую очередь мы создаем объект с именем `myPoll`, затем используем метод `DisplayPoll()` для вывода опроса на экран (мы будем рассматривать этот метод немного позже), затем регистрируем поданный голос посредством метода `Vote()` класса `Pcll` и, наконец, повторно обращаемся к методу `DisplayPoll()` для вывода состояния опроса. Если мы все сделали правильно, то счетчик голосов для второго ответа после этого должен увеличиться.

`DisplayPoll()` — это простой частный метод, который получает объект `Poll` и выводит его в выходной поток консоли. Он показывает собственно вопрос, а также все возможные ответы вместе с числом голосов, которые были поданы за каждый из них:

```
private static viod DisplayPoll (Poll aPoll)
{
    Console.WriteLine ("==Poll++");
    Console.WriteLine (aPoll.Question);
    for (int i=1; i<=(aPoll.AnswerCount); i++)
    {
        Console.Write (aPoll.Answers [i] + ": ");
        Console.WriteLine (aPoll.Votes [i] + " votes]}
    }
    Console.WriteLine ("==========");
}
```



Метод `DisplayPoll()` использует свойство `Question` для вывода текста вопроса, после чего он проходит по всем элементам в свойствах `Answers` и `Votes` и выводит текст ответа и число голосов для всех возможных вариантов ответа на данный вопрос.

Создав класс `Poll`, мы вправе ожидать, что выходной поток консольного приложения будет иметь вид, представленный на рисунке слева.

Структуры данных (уровень данных)

У нас имеется тестирующая программа, с помощью которой можно убедиться в правильности работы нашего кода. Теперь рассмотрим вопрос о том, каким образом мы собираемся хранить данные в XML-файле. Следует иметь в виду, что поскольку между классом `Poll` и данными опроса существует взаимно однозначное соответствие, то, в зависимости от используемого для взаимодействия с классом `Poll` интерфейса (будь то консольное приложение, Web Forms или Windows Forms), есть возможность для перехода впоследствии к другой архитектуре хранения данных. При таком использовании XML существует целый ряд ограничений, о которых мы поговорим несколько позже.

Каждый опрос состоит из нескольких видов данных:

- Question (строка)

- Множество Answers (строки)
- Множество счетчиков Votes (целые)

Отсюда следует возможность создать простой XML-файл (`polls.xml`) в корневой директории диска `C:\`. Файл будет выглядеть следующим образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<Poll Question="What development language do you primarily use?">
   <Answer Text="C#" Votes="10" />
   <Answer Text="C++" Votes="2" />
   <Answer Text="Java" Votes="34" />
   <Answer Text="JavaScript" Votes="12" />
   <Answer Text="Visual Basic" Votes="42" />
</Poll>
```

Обратите внимание на то, каким образом используется тег `Poll`, в котором содержится вся информация об опросе и хранится атрибут, представляющий вопрос данного опроса. В тег `Poll` вложена последовательность, состоящая из нескольких тегов `Answer`, в каждом из которых содержится текст ответа и счетчик голосов.

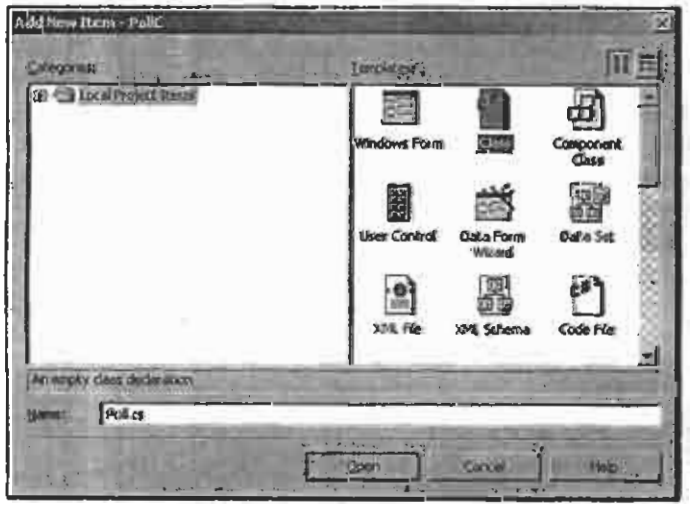
XML-формат обладает достаточно простой структурой, позволяющей выполнять редактирование вручную, но еще более ценным оказывается тот факт, что, сохранение данных в XML-формате предоставляет целый ряд возможностей для считывания данных и выполнения над ними манипуляций. Теги в XML-файле, где хранятся ответы на опрос, обладают точно такой же структурой, как если бы они хранились в базе данных — по одной строке на каждый ответ со столбцами `Text` и `Votes`. Как вы сможете увидеть впоследствии, мы будем осуществлять доступ к XML посредством набора данных точно таким же образом, как если бы мы осуществляли доступ к таблице, хранящейся в базе данных.

Реализация объекта

На уровне бизнес-логики используется единственный класс — класс `Poll`. Ниже перечислены методы и свойства, которые необходимо написать.

Методы

- `public static Poll Current()`
- `public void Vote(int answer)`



Свойства

- `public int AnswerCount`
- `public string Question`
- `public string[] Answers`
- `public int[] Votes`

Приступим к созданию класса. Прежде всего создадим пустой файл для класса `Poll`. Для этого в Visual Studio.NET нужно выбрать пункт **Add Class** меню **Project** и ввести имя файла `Poll.cs`, как показано на рисунке слева.

Visual Studio.NET создаст файл Poll.cs и сгенерирует общую схему класса, которая выглядит следующим образом:

```
using System;

namespace Poll
{
   /// <summary>
   /// Общее описание для Poll.
   ///</summary>
   public class Poll
   {
      public Poll ()
      {
         //
         // Действие: добавить здесь логически конструктор
         //
      }
   }
}
```

Обратите внимание на то, как используется пустой конструктор (метод Poll()); его использование отлнчается от файла Poll1c.cs, в котором вместо конструктора использовался метод Main(). Теперь следует пройти по всем методам и посмотреть, каким образом их можно реализовать.

Прежде чем переходить к рассмотрению методов и свойств класса Poll, необходимо определить некоторые поля. Более подробно эти поля будет рассматриваться для каждого конкретного метода. В полях будут храниться число голосов, ответы, вопрос, количество ответов (myAnswerCount) и полное имя XML-файла с данными об опросе:

```
namespace Polling
{
   public class Poll
   {
      private int [] my Votes;
      private string [[] myAnswers;
      private string myQuestions;
      private int myAnswerCount;
      private string myFilename;
```

Заметьте также, что мы изменили пространство имен на Polling.

Метод public static Poll Current()

Этот метод — единственный статический метод, он используется для получения экземпляра класса Poll с данными активного опроса, которые этот класс должен извлечь из XML-файла. Данный метод извлекает текст вопроса из тега <Question> в XML-файле и записывает его в свойство myQuestion. После этого он проходит в цикле по всем тегам <Answer> и переписывает значение Text в массив myAnswers, а значения Votes — в массив myVotes.

Поскольку метод предназначается для создания нового экземпляра класса Poll, то автор предпочел использовать для создания объекта Poll на основе XML-файла частный конструктор, в результате чего метод Current() становится очень простым — выполнение всей работы берет на себя конструктор:

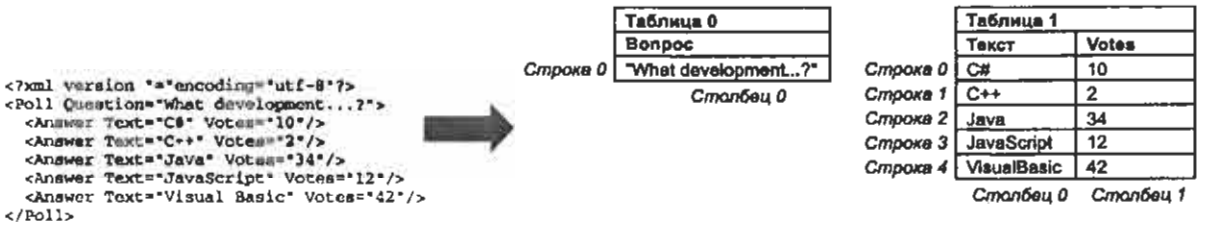
```
public static Poll Current()
{
   Poll myPoll = new Poll(@"C:\polls.xml");
   return myPoll
}
```

Здесь происходит только создание нового экземпляра `Poll` при помощи ключевого слова `new` и передача конструктору строки с именем файла. Обратите внимание на то, что мы добавляем символ '@' в начало строки, что позволяет использовать в названии файла символ обратного слэша (в противном случае пришлось бы использовать следующую запись `"c:\\polls.xml"`; применение двойного обратного слэша может оказаться очень громоздким). После создания нового объекта мы просто возвращаем его из метода. Теперь приступим к рассмотрению нового конструктора.

Метод private Poll(string filename)

Конструктор, который принимает имя файла в качестве своего единственного параметра, используется при создании объекта `Poll` и конфигурировании его посредством данных в XML-файле. Есть несколько возможностей извлечению данных из XML-файла. Можно либо последовательно каждый узел и извлекать необходимые данные, либо считывать только определенные участки данных, которые требуются (например, сначала строку `Question`, затем все строки `Answer` и т. д.). Есть и другая возможность — сериализовать объект `Poll` на диск, а затем десериализовать его и тем самым воссоздать объект в памяти. Существует и четвертая возможность: считать XML-файл в объект `DataSet`, что позволит обращаться с ним так же, как и с любым другим источником данных. Для нашего примера мы выбираем последний путь — считываем XML-файл в объект `DataSet` с помощью ADO.NET.

На приведенной ниже диаграмме видно, что при загрузке XML-файла в набор данных ADO.NET создаются две таблицы (что в объектах `DataSet` может храниться сразу несколько связаных таблиц). Таблица с номером ноль представляет тег `<Poll>` и содержит всего одну строку и один столбец; в клетке этой таблицы содержится вопрос текущего опроса. Таблица с номером один содержит в себе данные тега `<Answer>`; в ней хранятся все ответы опроса в виде двух столбцов, по одной строке на каждый ответ:



Пройдем последовательно по всему методу шаг за шагом, создавая код для загрузки XML-файла в набор данных. Нам придется написать здесь еще несколько строе кода по сравнению с тем, что написали до сих пор. Сначала мы определяем сигнатуру конструктора, в которую входит строка с именем XML-файла с данными опроса:

```
private Poll(string filename)
{
   myFilename = filename;
```

Затем создаем объект `DataSet`, в котором будут храниться считываемые данные. Объекты `DataSet` обсуждались в главе 19. Мы создаем экземпляр объекта `FileStream` с помощью конструктора, которому передается как имя файла, так

и значение перечислимого типа FileMode.Open, следовательно, требуется открыть данный файл для чтения:

```
DataSet myDataSet = new DataSet();
FileStream fsReadXml = newFileStream(filename, FileMode.Open);
```

Чтобы оба этих объекта могли быть использованы, следует убедиться в том, что в самом начале файла присутствуют ссылки на соответствующие пространства имен:

```
using System.Data;              // Требуется для объекта DataSet
using System.IO;                // Требуется для объекта FileStream
```

Теперь переходим к созданию объекта XmlTextReader, который будем использовать для считывания данных из XML-файла. Для этого просто передаем созданный нами объект типа FileStream — fsReadXml:

```
XmlTextReader myXmlReader = new System.Xml.XmlTextReader(fsReadXml);
```

Не забудьте сослаться на следующее пространство имен:

```
using System.Xml;               // Для объекта XmlTextReader
```

Следующая строка кода выполняет за нас очень большой объем работы — в ней осуществляется вызов метода ReadXml() на объект DataSet. Этот метод считывает весь XML-файл в объект DataSet, а также создает таблицы и строки, необходимые для точного отображения не только самих данных, но и их структуры. ADO.NET формирует две таблицы; первая из них (таблица 0) состоит всего из одной строки и из одного столбца и содержит вопрос для опроса. Вторая таблица содержит по одной строке на каждый возможный ответ и два столбца — один для текста ответа, а второй для числа голосов, которые этот ответ получил:

```
myDataSet.ReadXml(myXmlReader);
```

Следующая строка выполняет закрытие файла:

```
myXmlReader.Close();
```

Остальная часть метода используется для извлечения данных, находящихся внутри набора данных, и переноса их в частные поля класса Poll. Прежде всего необходимо определить количество строк в таблице 1 (помните, что таблицы в наборах данных индексируются с нуля), чтобы узнать, какое количество ответов допускается при данном опросе, а затем сохранить это значение в поле myAnswerCount класса Poll:

```
myAnswerCount = myDataSet.Tables[1].Rows.Count;
```

Чтобы присвоить значение полю myQuestion (доступ к которому осуществляется через свойство Question класса Poll), мы обращаемся к нулевой таблице и выбираем первый элемент в первой и единственной содержащейся в ней строке. Свойство ItemArray возвращает объект типа object, поэтому необходимо воспользоваться методом ToString() (который предусмотрен в типе object), чтобы преобразовать объект в строку перед присваиванием полю myQuestion:

```
myQuestion = myDataSet.Tables[0].Rows[0].ItemArray[1].ToString();
```

В следующем блоке кода инициализируется массив myAnswers, предназначенный для хранения возможных ответов, а он наполняется проходом в цикле по всем строкам таблицы 1, в которой содержатся эти ответы. Поскольку мы используем массив с индексом, отсчет которого начинается с единицы (а не с нуля), то при

указании размера массива используется выражение `myAnswerCount+1`. Свойства `Votes` и `Answers`, описанные в классе `Poll`, также индексируются, начиная с единицы, поскольку такой интерфейс оказывается более понятным для уровня представления, ибо позволяет называть "вопросом 1" первый вопрос, а не второй (как это было бы при отсчете индекса с нуля). Это субъективное решение автора — вы можете внести изменения в код и сделать индекс массива, изменяющимся с нуля.

Обратите внимание на то, как мы осуществляем доступ к каждому ответу в таблице 1 с перечнем возможных ответов, с помощью прохода в цикле по всем строкам и извлечения каждый раз текста в нулевом столбце:

```
myAnswers = new string[myAnswerCount+1];
string answer;
for(int i=0;i<myAnswerCount;i++)
{
   answer = myDataSet.Tables[1].Rows[i].ItemArray[0].ToString();
   myAnswers[i+1] = answer;
}
```

Следующий блок кода практически идентичен предшествующему за исключением того, что на этот раз мы наполняем массив `myVotes` и используем для этого первый столбец таблицы. Поскольку данный массив содержит в себе значения целого типа, то приходится для объекта типа `int` использовать метод `Parse()`, который преобразует строку XML-файла в значение целого типа, прежде чем присвоить ее элементу массива `myVotes`:

```
myVotes = new int[myAnswerCount+1];
                    // прибавляем 1, поскольку индекс отсчитывается с 1
string votes;
for(int i=0;i< myAnswerCount;i++)
{
   votes = myDataSet.Tables[1].Rows[i].ItemArray[1].ToString();
   myVotes[i+1] = int.Parse(votes);
}
```

В этом конструкторе объект инициализирован таким образом, что полям `myQuestion`, `myAnswerCount`, `myAnswers` и `myVotes` присвоены соответствующие значения, и доступ к этим значениям может быть осуществлен использованием следующих свойств объекта: `Question`, `Count`, `Answers` и `Votes`. После создания объект `Poll` остается единственная (на данный момент) возможность вносить изменения в хранящиеся в нем данные, и она заключается в методе `Vote()`, к рассмотрению которого мы переходим.

Очень важно понимать, что при использовании свойства класса `Poll` мы, на самом деле, считываем данные не непосредственно из XML-файла, а из полей, находящихся внутри класса. Причина, по которой программа устроена именно так, заключается в том, что в этом случае не требуется открывать XML-файл при каждом обращении к какому-либо свойству класса. Такой подход носит название "кэширования" и позволяет существенно увеличить быстродействие класса.

Метод public void Vote(int answer)

Этот метод принимает в качестве единственного параметра целое значение, которое определяет, какой именно ответ был выбран. Далее он использует это значение, чтобы добавить один голос данному ответу как в поле `myVotes` класса `Poll` (в кэшированной копии), так и в самом XML-файле.

В первой строке, выполняемой в этом методе, осуществляется проверка значения, которое было передано методу параметром `answer`. Если это значение либо

равно нулю, либо превосходит количество ответов, предусмотренных для данного опроса (которое хранится в переменной myAnswerCount), то для создания исключительной ситуации используется ключевое слово throw. Это приведет к ошибке, которая будет выводиться пользователю, если в вызывающем коде не будет в явном виде предусмотрен ее перехват:

```
public void Vote(int answer)
{
   if (answer == 0 || answer > myAnswerCount)
   {
      throw (new Exception("Invalid choice of answer."));
                                        / Некорректный выбор ответа./
   }
```

Чтобы увеличить на единицу число голосов в кэшированной копии, которая хранится в массиве Votes, используется оператор ++ с элементом массива:

```
myVotes[answer]++;
```

Следующие несколько строк программы идентичны коду, расположенному недалеко от начала конструктора метода Poll: в нем проводится считывание XML-файла в объект DataSet:

```
DataSet myDataSet = new DataSet();
FileStream fsReadXml = FileStream(myfilename, System.IO>FileMode.Open);
XmlTextReader myXmlReader = new System.Xml.XmlTextReader(fsReadXml);
myDataSet.ReadXml(myXmlReader);
myXmlReader.Close();
```

Затем из DataSet извлекается строка с количеством голосов в ответе, номер которого определяется с помощью параметра answer, переданного этому методу. Очень важно быть максимально внимательным при создании данного приложения, поскольку все объекты DataSet индексируются, начиная с нуля, однако здесь для удобства используется индексы, отсчет которых начинается с единицы. Именно поэтому при указании строки в DataSet используется выражение "answer–1", которое позволяет перейти от одной системы индексирования к другой.

Напоминаем, что причина, по которой мы используем модификаторы индекса "+1" и "–1", заключается в том, что мы хотим, чтобы пользователь класса Poll имел возможность работать с вопросами, которые нумеруются 1, 2, 3, ..., а не 0, 1, 2, ... Хотя такой подход и облегчает жизнь тем, кто будет писать код уровня представления, он требует несколько большей работы. Этот вариант кода зачастую является источником ошибок, и его тестирование жизненно важно, поскольку обнаружение ошибок в такого рода программах оказывается весьма затруднительным:

```
string votes = myDataSet.Tables[1].Rows[answer-1].ItemArray[1].ToString();
```

После получения значения числа голосов оно преобразовывается в целый тип:

```
int votesInt = int.Parse(votes);
```

Чтобы изменить значение в ячейке таблицы в наборе данных, следует проникнуть внутрь него и извлечь объект DataRow, который представляет интересующий нас конкретный ответ на опрос. Поскольку индексация происходит внутри набора данных (отсчет с нуля), то индексатор модифицируется посредством добавления к нему "–1":

```
DataRow myVotesRow = myDataSet.Tables[1].Rows[answer-1];
```

Чтобы изменить значение переменной с количеством голосов, нужно добавить в нее единицу, преобразовать ее в строку и присвоить ее столбцу `Votes` объекта `DataRow`. После выполнения этой строки в наборе данных будет находиться измененное значение счетчика голосов, поэтому все, что теперь остается сделать,— это записать набор данных обратно в XML-файл.

```
myVotesRow["Votes"] = (votesInt+1).ToString();
```

Далее создается объект `StreamWriter`, в который будут записаны хранящиеся в объекте `DataSet` данные, и ему передается имя XML-файла, хранящегося в поле `myFilename`:

```
StreamWriter myStream = new StreamWriter(myFilename);
```

Чтобы передать ADO.NET информацию о том, что XML-содержимое должно быть включено обратно в файл, нужно обратиться к методу `WriteXml()` объекта `DataSet`, передавая ему соответствующий поток, а также значение, которое указывает на то, что мы хотим записать XML без использования информации, хранящейся в схеме:

```
myDataSet.WriteXml(myStream,XmlWriteMode.IgnoreSchema);
```

В заключение мы закрываем поток и завершаем метод:

```
    myStream.Close();
}
```

Это все, что необходимо, чтобы сохранить измененные данные. .NET позволяет перемещать данные из XML в наборы данных и обратно фантастически просто.

Свойство public int AnswerCount

Свойство `AnswerCount` возвращает количество ответов в опросе. Оно используется в режиме "только чтение" и возвращает значение, которое хранится в частном поле `myAnswerCount`. Это поле инициализируется исключительно с помощью метода `Current()`, хотя последующие версии класса могут допускать создание экземпляров и работу с этим полем другими способами. Например, если мы захотим создавать новые варианты опроса программным путем, то нам потребуется возможность создавать новый опрос и настраивать число возможных в нем ответов:

```
public int AnswerCount
{
    get
    {
        return myAnswerCount;
    }
}
```

Свойство public string Question

Это свойство полностью идентично свойству `AnswerCount` за исключением того, что оно возвращает значение поля `myQuestion`, в котором хранится строка с вопросом, задаваемом при проведении данного опроса:

```
public int Question
{
    get
    {
        return myQuestion;
    }
}
```

Свойство public string[] Answers

Это свойство возвращает массив фиксированной размерности, в котором хранится текст для всех ответов, предусмотренных данным опросом. Для этого им возвращается частное поле `myAnswers`, которое было наполнено с помощью метода `Current()`:

```
public string[] Answers
{
   get
   {
      return myAnswers;
   }
}
```

Свойство public int[] Votes

Это свойство возвращает массив `myVotes`, в котором хранятся счетчики голосов для всех ответов данного опроса. Поле `myVotes` также наполняется с помощью метода `Current()`:

```
public int[] Votes
{
   get
   {
      return myVotes;
   }
}
```

Здесь необходимо принимать во внимание тот факт, что данное свойство возвращает значение счетчика голосов из кэшированной копии, а не в результате регулярного считывания этого значения из XML-файла, поэтому изменение значений счетчиков голосов, производимое более, чем одним объектом `Poll`, приведет к возникновению проблем. Если потребуется предусмотреть возможность регистрации голосов с помощью сразу нескольких объектов `Poll`, то целесообразнее реализовать это свойство заново таким образом, чтобы оно каждый раз считывало счетчики голосов непосредственно из источника данных.

Пространства имен

Для придания нашему примеру законченного вида и в качестве напоминания о используемых пространствах имен .NET Framework, перечислим еще раз те из них, которые должны быть включены в класс `Poll`:

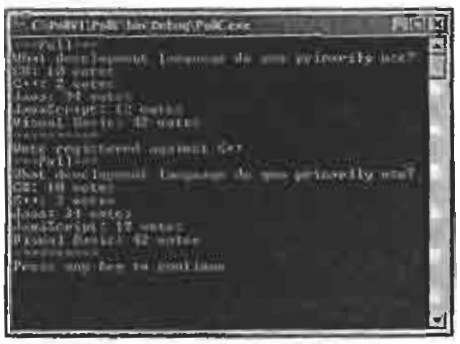
```
using System;
using System.Data;
using System.IO;
using System.Xml;
```

Запомните, что если вы включите объявления пространств имен до написания основного кода, то Visual Studio.NET окажет вам помощью в написании кода с помощью программы `IntelliSense`.

Тестирование класса Poll

Ранее нами было создано консольное приложение, которое может быть использовано для тестирования бизнес-объекта `Poll`. Теперь необходимо построить класс `Poll`, затем заново построить консольное приложенне, включив в файл, в котором

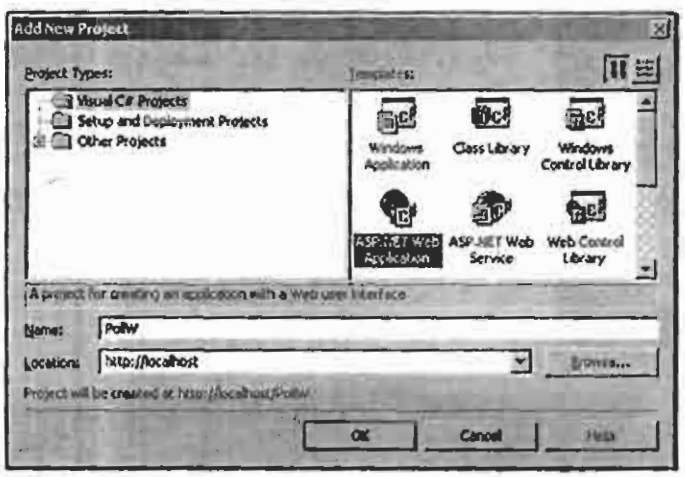
хранятся классы приложения, оператор `using Polling;`, а также сохранить наш XML файл в корневой директории диска `C:\` и добавить ссылку на модуль `Poll` с помощью Visual Studio.NET. Чтобы построить проект в виде решения в Visual Studio.NET, надо просто выбрать пункт **Build Solution** из меню **Build** (*Ctrl+Shift+B*). Для запуска консольного приложения нужно щелкнуть правой кнопкой мыши на проекте `PollC` и выбрать пункт **Set As Startup Project**, а затем выбрать пункт **Start Without Debugging** в меню **Debug** (*Ctrl+F5*). В результате мы увидим выходной поток, показанный на рисунке справа.



Эта тестирующая программа является относительно простой и не позволяет проверить работу всех аспектов бизнес-объекта, однако она оказывается вполне удобной для общей проверки. Одним из моментов, проверка которого не производится, является сохранение данных на диске, а также то, попадают ли реальные данные в XML-файл.

Многие программисты пишут тестирующий код, подобный этому еще до того, как они приступают к программированию бизнес-объекта, и убеждаются, что такая программа позволяет осуществлять тщательное тестирование бизнес-объектов. Это означает, что по мере внесения изменений в свой собственный код вы сможете легко обнаружить сделанные ошибки.

Пользовательский интерфейс (уровень представления)



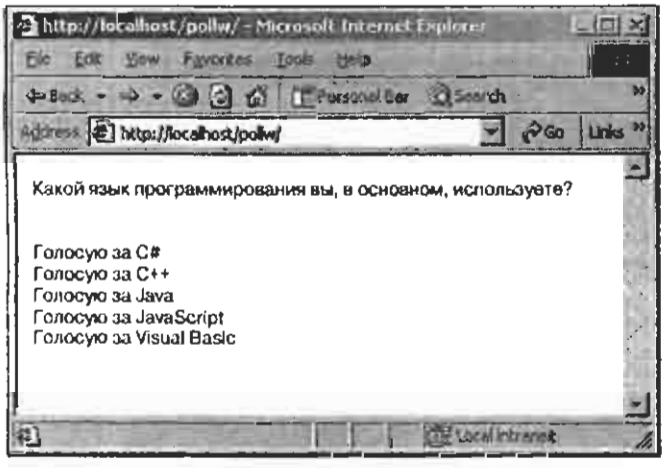
Завершив создание уровня данных и уровня бизнес-логики и поэкспериментировав с консольным приложением на уровне представления, можно перейти к созданию web-интерфейса для опроса с использованием ASP.NET. Прежде всего необходимо добавить проект ASP.NET в наше решение в Visual Studio.NET. Назовем его `PollW` (сокращение от Poll Web), как показано на рисунке слева.

Заметьте, что при создании нового web-приложения Visual Studio.NET автоматически генерирует несколько файлов, включая файл `WebForm1.aspx` — начальную страницу ASP.NET. Переименуем ее в `default.aspx` с тем, чтобы можно было ссылаться на приложение следующим образом: `http://localhost/PollW`. У файла `default.aspx` имеется привязанный к нему файл `default.aspx.cs` с кодом поддержки. Мы скоро приступим к работе с ним. Совершенно очевидно, что если web-сервер выполняется не на вашей локальной машине, то необходимо указать имя вашего сервера, а не использовать слово `localhost`.

Но прежде необходимо подумать об одном важном моменте. Использование ASP.NET позволяет создать фантастическую систему для построения web-форм, которая во многом напоминает Visual Basic, поскольку в Visual Studio.NET можно переносить управляющие элементы с помощью мыши и использовать программирование, основанное на работе с событиями. Однако существует одна загвоздка — при использовании ASP.NET можно использовать только одну форму на стороне

пользователя для каждой конкретной страницы. При создании web-сайта для проведения опросов представляется логичным воспользоваться на стороне сервера формой, которая будет обладать, например, одним переключателем для каждого ответа и кнопкой Vote для регистрации сделанного выбора. Если остановиться на таком подходе, то страница, на которой будет проводиться опрос, не сможет содержать никаких других форм на стороне сервера, а если включить опрос на несколько страниц web-сайта, где впоследствии потребуется добавить еще какие-либо формы (например, второй опрос или управляющий элемент поиска), то этого сделать будет нельзя. Чтобы избежать проблем подобного рода, а заодно получить возможность познакомиться с некоторыми не столь широко известными методами .NET, нужно воспользоваться такой моделью web-сайта для проведения опросов, в которой не будет использовано никаких форм на стороне сервера, т. е. перейти не более низкий уровень возможностей ASP.NET, который может оказаться знакомым тем, программирует на ASP 3.0. И хотя использование этих возможностей в ASP.NET является несколько необычным, все равно будет полезно познакомиться с тем, как они работают в подобных ситуациях — когда использование формы на стороне сервера оказывается неприемлемым.

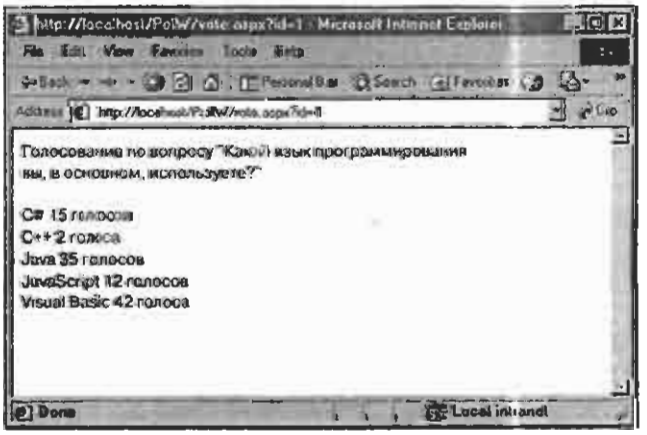
Обычно в простой, отсылаемой назад форме достаточно ограничиться одной страницей и воспользоваться событием `OnClick` для кнопки отправки (кнопки Vote), которая позволяет зарегистрировать выбор посетителя. Однако, чтобы избежать использования форм на стороне сервера, мы собираемся использовать две страницы. На первой из них будет выводиться вопрос и перечень возможных ответов, а рядом с каждым ответом — ссылка Vote (Голосую за). Эта ссылка позволяет передать в качестве параметра номер выбранного ответа второй странице, которая использует переданное ей значение для регистрации голоса, после чего выводит на экран текущие результаты голосования.



Как должна выглядеть первая страница — `default.aspx`, показано на рисунке слева.

А вот как должна выглядеть вторая страница — `vote.aspx`, представлено на рисунке справа.

Первая страница — `default.aspx` — представляет опрос, и когда мы выбираем первый из возможных ответов с помощью ссылки Vote, то видим, что странице `vote.aspx` в строке URL передается единица. Данный параметр используется для регистрации голоса. После этого `vote.aspx` выводит все варианты ответов со счетчиками голосов для каждого из них.



Начнем с файла ASP.NET `default.aspx`, созданного Visual Studio.NET. Эта страница является стандартной и в ней содержится несколько тегов, в частности, внешние HTML-теги, а также теги `<head>` и `<body>`:

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits=PollW.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
   <HEAD>
      <meta name ="GENERATOR" Content="Microsoft Visual Studio 7.0">
   </HEAD>
   <BODY>
      <form id="Default" method="post" runat="server">
      </form>
   </BODY>
</HTML>
```

Обратите вниманне на то, что в код также включен тег серверной формы (`<form runat="server">`). Первое, что нужно сделать,— удалить его; это послужнт напомнннанием того, что нами не используются формы на стороне сервера. В то же время мы включим пару управляющих элементов `Label` на стороне сервера; это будет сделано в том месте, где происходит вывод вопроса для опроса и перечня возможных ответов:

```
<%@ Page language="c#" Codebehind="Default.aspx.cs" AutoEventWireup="false"
Inherits=PollW.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
   <HEAD>
      <meta name ="GENERATOR" Content="Microsoft Visual Studio 7.0">
   </HEAD>
   <BODY>
      <asp:Label id="QuestionLabel" runat="server"></asp:Label>
      <asp:Label id="AnswerLabel" runat="server"></asp:Label>
   </BODY>
</HTML>
```

Теперь есть чистая страница ASP.NET с управляющим элемеитом `Label`, куда мы можем вывести наш вопрос. Под вопросом мы выведем возможные ответы, однако вместо того, чтобы делать это непосредственно в `.aspx`-файле, мы используем для этих целей файл с кодом поддержки — `default.aspx.cs`.

Здесь используется стандартный код, передаваемый Visual Studio.NET в файл `default.aspx.cs`, который осуществляет основную обработку событнй, а также содержит стандартные методы для инициализации и загрузки страницы. Обратите внимание на то, как Visual Studio.NET генерирует обработчик событий для метода `Page_Init()`; именно так объект `Page` вызывает метод `Page_Init()` после его создания:

```
using Polling;
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using Syset.Web;
using System.Web.SeesionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using SYstem.Web.UI.HtmlControls;
```

```
Namespace PollW
{
   /// <summary>
   /// Общее описание для WebForm1.
   /// </summary>
   public class webForm1 : System.Web.UI.Page
   {
      public WebForm ()
      {
         Page Init += new System.EventHandler (Page_Init);
      }

      private void Page_Load (object sender, System.EventArgs e)
      {
         // Ввод кода пользователя для инициализации страницы
      }

      private void Page_Init (object sender, EventArgs e)
      {
         //
         // Генерация кода: Этот запрос требуется ASP.NET
         //
         InitializeComponent ();
      }

      #region Web Form Designer geenrated code
      /// <summary>
      /// Необходимый метод для поддержки Design - не модифицировать
      /// содержание этого метода с помощью редактора кода.
      /// </summary>
      private void InitializeComponent ();
      {
         this load += new System.EventHandler (this.Page_Load);
      }
      #endregion
   }
}
```

Первое изменение, которое необходимо внести, заключается в добавлении объявлений для `QuestionLabel` и `AnswerLabel`, что позволит использовать их из кода поддержки страницы. Если вы используете Visual Studio.NET и переносили управляющие элементы `Label` мышью, а не добавляли их вручную, то все уже сделано за вас. В самом верху класса `WebForm1`, непосредственно перед конструктором должны появиться следующие строки:

```
public class WebForm1 : System.Web.UI.Page
{
   protected System.Web.UI.WebControls.Label QuestionLabel;
   protected System.Web.UI.WebControls.Label AnswerLabel;
```

Основное внимание мы уделим методу `Page_Load()`, который вызывается в тот момент, когда страница запрашивается браузером. Внутрь этого метода мы поместим код, отвечающий за вывод текста вопроса, и добавим управляющие элементы HTML для каждого из возможных ответов. Давайте пройдем по данному методу шаг за шагом и разберем новый код. Сначала мы создаем объект класса `Poll` и инициализируем его используя статический метод `Current()`:

```
private void Page_Load(object sender, System.EventArgs e)
{
   Poll aPoll = Poll.Current();
```

Использование объекта `Poll` начинается в следующей строке; он используется для выполнения присваивания значения свойству `Text` управляющего элемента Label (`QuestionLabel`):

```
QuestionLabel.Text = aPoll.Question;
```

Несколько последующих строк предназначаются для организации цикла, который проходит по всем возможным ответам и выводит для каждого из них некоторый HTML. Обратите внимание на то, что для задания числа повторений цикла `for` используется свойство `Poll.AnswerCount`. Мы также создаем строковую переменную, в которой будет храниться создаваемый нами HTML, и вставляем в нее два тега, ответственных за переход на новую строку:

```
string HtmlLink = "<br /><br />";
for(int i=1;i<=aPoll.AnswerCount;i++)
{
```

Для каждого из ответов требуется вывести текстовую ссылку с текстом "Vote" ("Голосую за"), а также текст самого ответа. Нужно внимательно следить за тем, чтобы ссылка действительно указывала на `vote.aspx`, а в значение URL включалось число, которое соответствует номеру выбранного ответа (например, при выборе ответа "C#" — это 1). HTML-код будет выглядеть следующим образом:

```
<a href="vote.aspx?id=1">Vote</a> C#<br />
```

Обратите также внимание на использование в конце строки тега `<br />`, который отвечает за то, чтобы каждый ответ на экране располагался на отдельной строке. Для того чтобы более аккуратно отформатировать вывод, можно бы воспользоваться таблицей, но пока мы не будем ничего усложнять.

Запись `" "` расшифровывается как non-breaking space (неудаляемый пробел). Она заставляет браузер вставить пробел между текстом `"Vote"` и текстом самого ответа (например, C#), т. е. эти две записи при выводе будут разделены строго одним пробелом. Для создания соответствующего HTML-кода нужно использовать некоторую строку (с именем `HtmlLink`), а также переменную `i` и свойство с массивом `Answers` объекта `Poll`, для того чтобы вставить значение `id` и текст ответа. Код разбивается на две строки, поэтому он умещается на экране (обычно, если создаваемая текстовая строка занимает несколько строк на экране, следует воспользоваться классом `StringBuilder`):

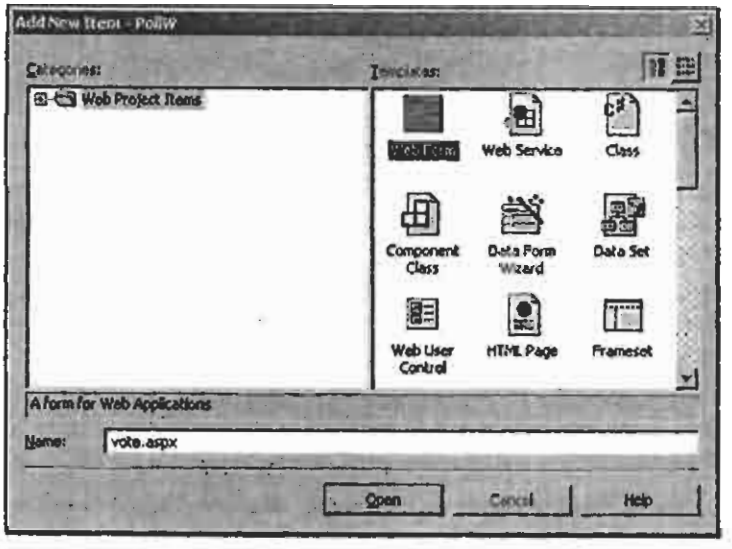
```
HtmlLink += "<a href=\"vote.aspx?id="+i+"\">Vote</a>";
HtmlLink += " " + aPollAnswers[i] + "<br />";
```

Теперь есть строковая переменная — `HtmlLink`, которая содержит HTML-код для одного из ответов. Далее следует перейти к очередной итерации цикла и добавить HTML-текст для следующего ответа, а потом присвоить текст второму управляющему элементу `Label` с помощью HTML-кода:

```
    }
    AnswerLabel.Text = HtmlLink;
  }
}
```

Если теперь загрузить `default.aspx` в браузер, то можно увидеть созданный нами опрос, с выведенным в самом верху вопросом, под которым располагается перечень ответов (но не забудьте, что ссылки Vote пока не работают).

Теперь нужно перейти к странице `vote.aspx`. Для ее создания VS следует воспользоваться пунктом Add Web Form меню Project в Visual Studio.NET. Убедившись,



что в окне диалога выбран пункт Web Form, введем vote.aspx в поле Name (см. рис. слева).

Visual Studio.NET сгенерирует оба файла — `vote.aspx` и `vote.aspx.cs` (файл с кодом поддержки), а также откроет файл `vote.aspx` для редактирования. Файл `vote.aspx` практически идентичен `default.aspx`; он представляет собой чистую страницу, содержащую управляющий элемент `Label`, в который будет помещен текст, представляющий данную страницу. Обратите внимание на то, что теги форм серверной стороны были удалены вручную за ненадобностью:

```
<%@ Page language="c#" Codebehind="vote.aspx.cs" AutoEventWireup="false"
      Inherits=PollW.vote" %>
      <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
      <HTML>
         <HEAD>
            <meta name ="GENERATOR" Content="Microsoft Visual Studio 7.0">
            <meta name ="CODE_LANGUAGE" Content="C#">
            <meta name ="vs_defaultClientScript" content="JavaScript (ECMAScript)">
            <meta name ="vs_targetSchema"
                  content="http://schemas.microsoft.com/intellisense/ie5- ">
         </HEAD>
         <BODY>
            <asp:Label id="titleLabel" runat="server"></asp:Label>
            <br />
            <br />
            <asp:Panel id="resultsPanel" runat="server"></asp:Panel>
         </BODY>
      </HTML>
```

Точно так же, как и для `default.aspx`, необходимо добавить для управляющих элементов следующие объявления:

```
public class vote : System.Web.UI.Page
{
   protected System.Web.UI.WebControls.Label titleLabel;
   protected System.Web.UI.WebControls.Panel resultsPanel;
```

Теперь следует приступить к рассмотрению метода `Page_Load()`. Он должен вставить название страницы, которая будет выполнять повторный просмотр вопроса текущего опроса, а затем выводить все ответы с указанием числа голосов, которые удалось собрать каждому из них. Сначала мы создаем экземпляр объекта `Poll` посредством метода `Current()`, в результате чего получается объект, в котором содержатся данные, касающиеся активного в настоящий момент опроса, взятые из XML-файла:

```
private void Page_Load(object sender, System.EventArgs e)
{
   Poll aPoll = Poll.Current();
```

В следующей строке мы используем объект `Request` класса `Page` (от которого наследуется данная страница) и с помощью семейства `QueryString` получаем доступ к параметрам, хранящимся в URL в строке запроса:

```
string parameter = Request.QueryString["id"];
```

Далее нужно произвести проверку того, не равно ли значение `id` нулю. Если это значение не нуль, то следует делать попытку зарегистрировать поданный голос; после чего в любом случае перейти к выводу на экран текущих результатов опроса. Обратите внимание на то, как мы опять используем метод `Parse()` класса `int` для преобразования строки в целое значение. Этот метод завершится неудачей, если ему будет передана строка, которая не может быть преобразована в целое значение, однако в данном случае уровень контроля за тем, что именно передается странице, очень высок, поэтому едва ли имеет смысл пытаться перехватывать такую ошибку:

```
if (parameter != null)
{
   aPoll.Vote(int.Parse(parameter));
}
```

В следующей строке программы осуществляется присваивание свойства `Text` управляющему элементу `Label` для осуществления перехода к странице, где содержится вводный текст данной страницы, в котором вопрос задается повторно. Обратите внимание на то, как в строке используются обратные слэши ('\'), чтобы включить в состав текста символы "кавычки" '"':

```
titleLabel.Text = "Votes for \"" + aPoll.Question + "\"";
```

Оставшаяся в методе `Page_Load()` часть кода предназначается для вывода на экран результатов опроса, при этом каждый из вариантов ответа выводится на своем обычном месте, а рядом с ним выводится число голосов, которое он собрал. Сначала в нем создается экземпляр объекта `Table` (который представляет собой HTML-таблицу с результатами голосования), затем используется цикл `for`, в котором просматриваются все ответы, а в качестве ограничителя используется значение свойства `AnswerCount`:

```
Table myTable = new Table();
for(int i=1;i<=aPoll.AnswerCount;i++)
{
```

После этого следует свести воедино HTML-таблицу (с помощью `myTable`) и информацию, хранящуюся в объекта `aPoll`. Мы создаем новую строку и добавляем ее в таблицу:

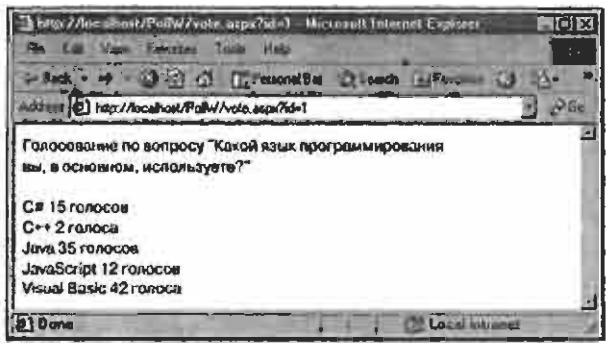
```
   TableRow myTableRow = new TableRow();
   myTable.Controls.Add(myRow);
```

Затем нужно создать новую ячейку таблицы, в которой свойство `Text` будет содержать результат опроса, и добавить эту ячейку в таблицу:

```
   TableCell myAnswerCell = new TableCell();
   myAnswerCell.Text = aPoll.Answers[i];
   myRow.Controls.Add(myAnswerCell);
```

После следует осуществить те же действия для счетчика голосов, для чего создать новую ячейку таблицы, присвоить ее свойству `Text` значение счетчика голосов (присоединяя в конец строки текст `"votes"`), и добавляя эту ячейку в строку таблицы:

```
            TableCell myVotesCell = new TableCell();
            myVotesCell.Text = aPoll.Votes[i].ToString() + " votes";
            myRow.Controls.Add(myVotesCell);
        }
```



После завершения цикла таблица добавляется в семейство управляющих элементов, хранящихся в серверном управляющем элементе `resultsPanel`, следующим образом:

```
    resultsPanel.Controls.Add(myTable);
}
```

В результат этой операции проиллюстрирован на рисунке слева.

Оценка версии 1

На настоящий момент мы являемся обладателями небольшого трехзвенного приложения, которое собирает ответы, полученные при проведении опроса, при этом мы имеем возможность обращаться к бизнес-объекту как из приложения ASP.NET, так и из консольного приложения. Наше приложение реализовано только в самых общих чертах, и поэтому представляется важным понимать присущие ему ограничения:

- **Многократное голосование.** Это, вероятно, является одним из самых существенных недостатков; каждый может зайти на сайт и многократно голосовать, отвечая на один и тот же вопрос опроса. Если результаты опроса являются значимыми для какой-либо третьей стороны, то вряд ли ее представители смогут устоять перед искушением добавить пару-тройку лишних голосов к тому или иному ответу.
- **Возможность повторного использования.** Хотя разработанная программа является самодостаточной, и при попытке ее повторного использования не должно возникать никаких трудностей, ее интерфейс оказывается распределенным между двумя файлами — с расширениями `.aspx` и `.cs` (файл с кодом поддержки). Каждый раз, когда возникает необходимость включить новый опрос, приходится объединять эти два файла с `.aspx`-файлом и `.cs`-файлом страницы, на которой предполагается проведение этого опроса. Было бы желательно иметь более простой способ включения опроса в страницу, и в ASP.NET такая возможность предусмотрена.
- **Редактирование опроса.** Несмотря на то, что XML-файл обладает весьма простой структурой, всегда существует определенная вероятность того, что в какой-то момент человек, занимающийся его редактированием, забудет какой-либо жизненно важный символ, и в выводе web-сайта начнут появляться ошибки. Справиться с такой ситуацией можно несколькими способами. Можно изящно обойти такую проблему и вывести сообщение типа "Извините, данный опрос в настоящий момент недоступен". Но более правильным решением будет предоставление менеджеру сайта интерфейса, с помощью которого он сможет выполнять редактирование опроса. Такой интерфейс мог бы быть реализован

с помощью целого ряда технологий, однако наиболее подходящими представляются либо ASP.NET (web-интерфейс, предназначенный для редактирования опроса), либо интерфейс Windows Forms.

- **Слежение за ходом опроса.** В том виде приложения, в котором оно существует на данный момент, не предусмотрено никакого простого способа, с помощью которого менеджер сайта мог бы ознакомиться с текущим состоянием опроса. Если собираемые данные представляют определенную важность, то было бы полезно иметь возможность наблюдать за распределением голосов в режиме реального времени. Для решения этой проблемы целесообразно создать дополнительный управляющий интерфейс. Такой интерфейс можно построить с использованием ASP.NET или — в качестве варианта — расширить уровень представления и создать интерфейс Windows Forms.
- **Управление данными.** Поскольку все данные хранятся в XML-файле, необходимо иметь возможность контролировать состояние этого файла. Поэтому при дублировании web-сервера XML-файл также должен дублироваться. Хотя выполнить это совсем несложно, дублирование требует дополнительных административных усилий, направленных на сопровождение web-сайта, особенно если часть данных хранится в других местах (например, в БД или в каких-либо других файлах). Серьезные проблемы возникают также, если когда-либо в будущем встанет задача расширить web-сайт, с тем чтобы он мог выполняться более, чем на одном сервере; в этом случае можно было бы хранить файл с данными на одном сервере, к которому имеют доступ оба web-сервера, однако это простая проблема быстро становится сложнее, чем в идеальном случае. Поэтому, если принимать во внимание все проблемы, связанные с сопровождением, может оказаться целесообразным сложить все яйца в одну корзину и воспользоваться централизованным хранением всех данных в одной базе данных, вместо того чтобы хранить их в XML-файле.
- **Повторяющийся код.** В методе `Current()` есть участок кода, отвечающий за загрузку XML-файла в объект `DataSet`; это код затем дублируется в методе `Vote()`. В реальной программе такой участок кода должен быть оформлен в виде отдельного частного метода, которым смогут пользоваться и метод `Current()`, и метод `Vote()` для получения объекта `DataSet` из XML-файла. Избегайте повторяющихся участков кода любой ценой — и ваша жизнь как программиста станет существенно проще.
- **Множественные опросы.** Данная модель предназначена для проведения только одного опроса. В реальной ситуации можно предположить, что каждый из опросов со временем будет заменяться на некоторый новый, т. е. менеджеру сайта потребуется редактировать XML-файл для замены опроса. При необходимости может быть поставлена задача сохранения копии предыдущего опроса в архиве опросов. Объем ручного труда, который потребуется для решения всех этих задач, оказывается достаточно большим, и в реальной жизни потребуется усовершенствовать модель, чтобы все опросы автоматически заносились в архив в используемом хранилище данных, а у менеджера web-сайта имелся в распоряжении интерфейс, позволяющий ему добавлять новые опросы. При наличии нескольких опросов возникает также необходимость устанавливать, какой из них является в настоящий момент активным.

- **Одновременное обращение.** Для хранения данных опроса используется единственный XML-файл, и в этом случае возникает возможность того, что определенная часть голосов не будет зарегистрирована, поскольку некоторые попытки осуществить доступ к файлу могут производиться в момент регистрации другого голоса. Маловероятно, что такая проблема возникнет для сайта с низкой интенсивностью посещений, но, если придется регистрировать большое количество голосов, может потребоваться внести соответствующие изменения, которые позволят справляться с высокой интенсивностью голосования. Наиболее логичным способом решения этой проблемы является переход на использование сервера БД, например SQL Server или Oracle, устойчивых к одновременным обращениям такого рода.
- **Внешний вид.** Для обеих страниц опроса — `default.aspx` и `vote.aspx` — не разработано никакого внешнего представления. Для любого web-сайта необходимо прорабатывать внешний вид. В данном примере первостепенное внимание уделяется процессу разработки и используемым для этого технологиям, поэтому придание web-сайту надлежащего внешнего вида мы оставляем полностью на ваше усмотрение!
- **Обработка ошибок.** В существующем на настоящий момент виде данное приложение обладает определенными функциональными возможностями, однако в нем предусмотрены только самые основные возможности обработки ошибок. Проблема заключается в том, что если происходит что-либо непредвиденное, то приложение генерирует сообщения об ошибках, которые совершенно не следует выводить пользователям. Например, если XML-файл оказывается недоступен, то возникает исключительная ситуация. Прежде чем приложение может быть передано для коммерческого использования, оно должно быть расширено, чтобы защищать себя и пользователей от возможных проблем.

Теперь, когда у нас имеется представление о некоторых из тех областей, в которых мы можем усовершенствовать и улучшить приложение для проведения опросов, давайте приступим к созданию второй версии, в которую будут включены некоторые дополнительные функциональные возможности и преодолены два из перечисленных выше ограничений. Мы рассмотрим только две проблемы из перечисленных, однако и об остальных проблемах не следует забывать, если вы когда-либо соберетесь создавать аналогичную систему проведения опросов.

Версия 2

Программы не создаются, они вырастают. По мере того как ведется работа с некоторой версией какого-либо ПО, постоянно возникают планы по расширению и изменению этого программного обеспечения. Существует целый ряд усовершенствований, которые могут быть внесены в наше приложение. Во второй версии мы собираемся решить проблему двух ограничений и продолжить совершенствование уже созданного кода, уделяя пристальное внимание тому, куда вносятся изменения, в частности, касается ли это уровня представления, уровня бизнес-логики или уровня данных.

Дополнительные требования

Ниже представлены требования ко второй версии web-сайта для проведения опросов, которыми мы будем руководствоваться:

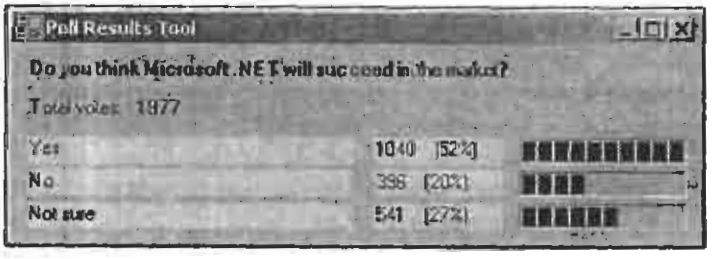
- **Инструмент для просмотра результатов опроса.** Менеджеру сайта необходима некоторая утилита, которую он мог бы запускать на своей рабочей машине и которая позволяла бы непрерывно выводить текущее состояние счетчиков голосов. Это приложение должно быть приложением Windows и устанавливать соединение с web-сайтом посредством TCP/IP. Этот инструмент должен выводить вопрос, все ответы, счетчик голосов для каждого из ответов и его долю в процентном отношении, а также суммарное количество голосов.
- **Один человек — один голос.** Посетители сайта должны иметь возможность проголосовать только один раз по каждому опросу.
- **Управляющий элемент для проведения опросов.** Должна быть предусмотрена возможность несложного добавления опроса на любую страницу ASP.NET просто за счет обеспечения доступа к бизнес-объекту и добавления в `.aspx`-файл нескольких простых операторов.

Инструмент для просмотра результатов опроса

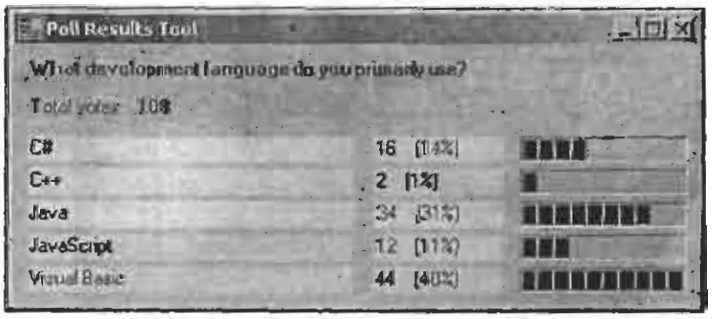
Интерфейс

Из трех новых требований к приложению для начала выбираем самое большое. Оно предполагает создание настольного приложения Windows, построенного с использованием Windows Forms. К нему не предъявляется никаких особых требований в смысле вывода информации об опросе, хотя представляется разумным предположить, что это будет происходить в рамках одной формы.

Модель, которой мы собираемся воспользоваться, состоит из нескольких надписей, располагающихся вверху, в которых содержится вопрос и суммарное количество проголосовавших. Под указанными надписями располагается группа строк, в каждой из которых содержится текст ответа и количество набранных голосов (вместе с процентным отношением), а также линейка, представляющая счетчик количества голосов в графическом режиме (см. рис. слева).



Все очень просто, за исключением одной маленькой загвоздки, которая заключается в том, что на момент разработки формы нам не известно общее количество голосующих. Можно было бы воспользоваться окном со списком либо каким-либо другим управляющим элементом, который в состоянии динамически изменять свои размеры, однако для того чтобы наш инструмент выглядел красиво, а также для того, чтобы продемонстрировать некоторые возможности Windows Forms, мы будем формировать каждую строку данных в динамическом режиме (см. рис. справа).



Это означает, что если в опросе оказывается большее количество ответов, то будет выведено большее количество строк и для их вывода форма изменит свои размеры.

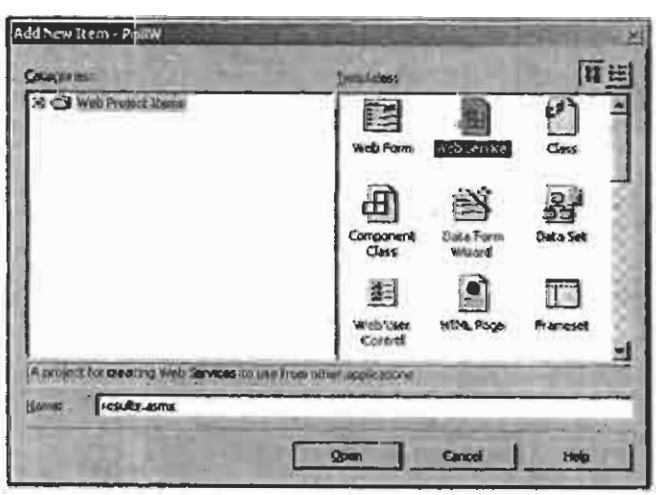
Web-служба для проведения опроса

Для реального сценария использования данного приложения представляется маловероятным, чтобы инструмент для просмотра результатов использовался на той же машине, на которой находится web-сайт. Напротив, с уверенностью можно утверждать, что он будет располагаться на другой машине, однако мы предполагаем, что соединение между этими двумя машинами поддерживает TCP/IP. Для того чтобы объекты могли взаимодействовать по каналу TCP/IP с использованием HTTP (что обычно предполагает работу в Интернете, поскольку интранет-приложения обычно используют работу с удаленными объектами), следует воспользоваться web-службами. Web-службы — это способ производить удаленные вызовы по различным сетям, включая Интернет, а также использовать SOAP- и XML-стандарты для управления соединением. При использовании .NET, в особенности Visual Studio.NET, реализовать web-службу не составляет никакого труда. При этом многое из происходящего будет скрыто от наших глаз, но само построение web-службы оказывается относительно простым.

Первое, что мы собираемся сделать,— это создать web-службу для проведения опросов; она будет создана в приложении ASP.NET PollW, поскольку обращения к этой web-службе будут происходить из страниц нашего web-сайта. В web-службе будут предусмотрены три web-метода, доступ к которым может осуществляться через Интернет:

- `Answers()`. Возвращает массив строк, в котором содержится текст всех ответов.
- `Votes()`. Возвращает массив целых, в котором содержатся счетчики голосов для каждого из ответов.
- `Question()`. Возвращает строку, содержащую вопрос текущего опроса.

Теоретически можно сделать эти методы доступными через web, описав их в классе `Poll`, однако, чтобы не смешивать методы, доступ к которым осуществляется через Интернет, с нашим бизнес-объектом, мы создадим новый класс, где будут находиться эти web-методы, которые, в свою очередь, используют методы, описанные в уже существующем классе `Poll`. В подходе, обеспечивающем полный контроль над интерфейсом web-служб, который не зависит от бизнес-логики класса более низкого уровня, имеются определенные преимущества.



Все, что требуется для создания web-службы в Visual Studio.NET,— это воспользоваться пунктом **Add Web Service** меню **Project**, позволяющим добавить web-службу в проект `PollW` (web-приложение). Далее следует ввести имя файла web-службы; это та страница, которая будет предоставлять наши методы. Web-службами используются файлы с расширением `.asmx`, хотя их структура и содержание почти не отличаются от файлов с расширением `.aspx` (см. рис. слева).

После того создания файла с расширением `.asmx` можно обнаружить, что кроме него создан еще один файл `results.asmx.cs` с кодом поддержки. Именно в нем будет храниться код, реализующий web-методы. Если вы прокрутите файл `results.asmx.cs` вниз, то ближе к его концу вы обнаружите образец web-метода с комментариями:

```
// ОБРАЗЕЦ WEB-СЛУЖБЫ
// Образец службы HelloWorld() осуществляет возврат строки Hello World
// Для создания web-службы уберите знаки комментария из следующих строк,
// затем сохраните проект и постройте его
// Для тестирования web-службы нажмите F5
//
//       [Webmethod]
//       public string HelloWorld()
//       {
//          return "Hello World";
//       }
```

Из приведенного выше примера видно, что все, что требуется для создания web-метода,— это включить атрибут "`[Webmethod]`" непосредственно перед описанием самого метода. Далее приводится код web-метода `Question()`:

```
[Webmethod(Description="Return the poll question")]
                                    / описание: возвращает вопрос опроса /
public string Question()
{
   Poll aPoll = Poll.Current();
   return aPoll.Question;
}
```

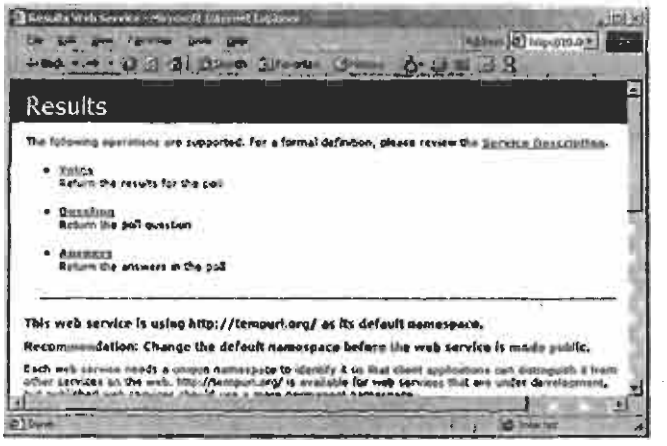
Обратите внимание на то, что атрибуту `[Webmethod]` передается некоторый параметр, в котором содержится краткое описание web-метода. В теле самого метода используется класс `Poll` точно таким же образом, каким он использовался в нашем консольном приложении и в web-приложении. Он создает экземпляр объекта класса `Poll` с помощью метода `Current()` и возвращает значение свойства `Question`. Весь код, необходимый для реализации возможности обращения к этому методу через Интернет, скрыт от нас.

Web-методы для счетчиков голосов и для ответов абсолютно аналогичны:

```
[Webmethod(Description="Return the results for the poll")]
                    / описание: возвращает результаты проведения опроса /
public int[] Votes()
{
   Poll aPoll = Poll.Current();
   return aPoll.Votes;
}

[Webmethod(Description="Return the answers in the poll")]
                        / описание: возвращает возможные ответы опроса /
public string[] Answers()
{
   Poll aPoll = Poll.Current();
   return aPoll.Answers;
}
```

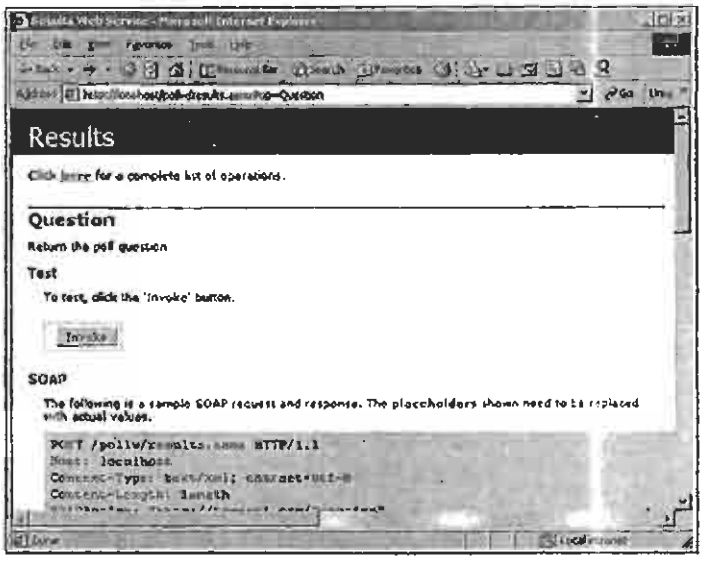
Для каждого из методов мы включили краткое описание; все эти методы используют класс `Poll` для предоставления требуемых данных. Именно в этом заключается все изящество централизованных бизнес-объектов; создав объекты, обладающие основными функциональными возможностями, мы затем можем легко использовать их в различных местах.



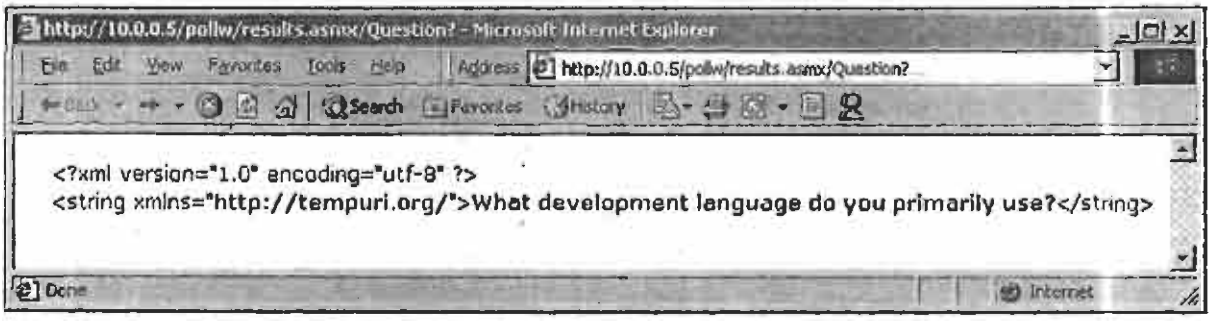
Если теперь открыть файл `results.asmx` в браузере через HTTP, то можно увидеть то, что изображено на рисунке слева. Обратите внимание на то, что при перечислении всех методов указывается не только его имя, но и краткое описание. Эта информация существенно облегчает понимание того, какие действия выполняет каждый из web-методов.

Эта страница в удобном для чтения виде автоматически генерируется системой .NET и позволяет проверить работу web-службы. На ней отображаются все три web-метода.

Остальная часть страницы содержит рекомендации, каким образом следует сконфигурировать web-службу, перед тем, как передать ее для коммерческого использования. Если выбрать web-метод `Question`, то можно увидеть страницу (см. рис. справа) с информацией о выбранном web-методе и о том, как его следует использовать.

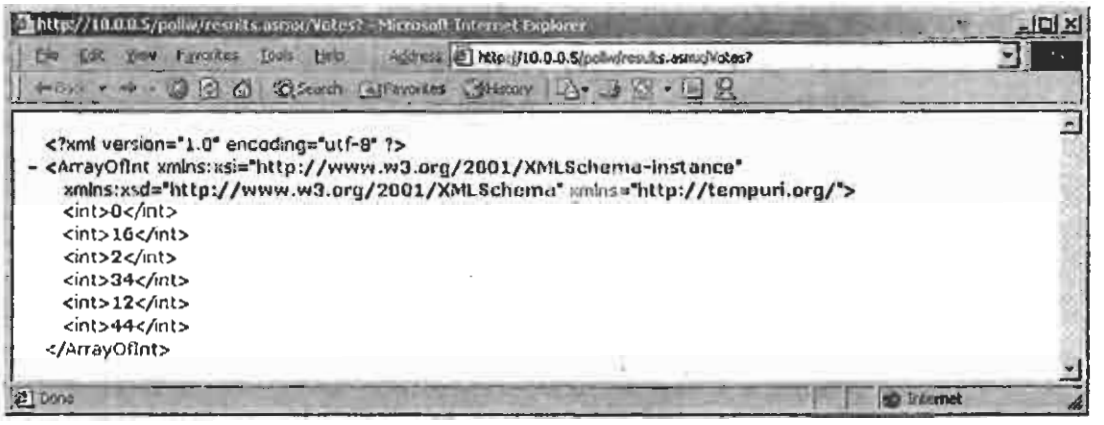


На странице web-метода `Question` расположена кнопка с текстом `Invoke` (вызов), которая может быть использована для тестирования web-службы. Если щелкнуть на кнопке, то можно будет увидеть выходной поток web-метода `Question()`, представленный на рисунке в виде SOAP/XML:



Заметьте, насколько проста строка URL, позволяющая обратиться к этому методу, а также на то, что результат работы метода — вопрос нашего опроса — возвращается в виде простого XML-документа.

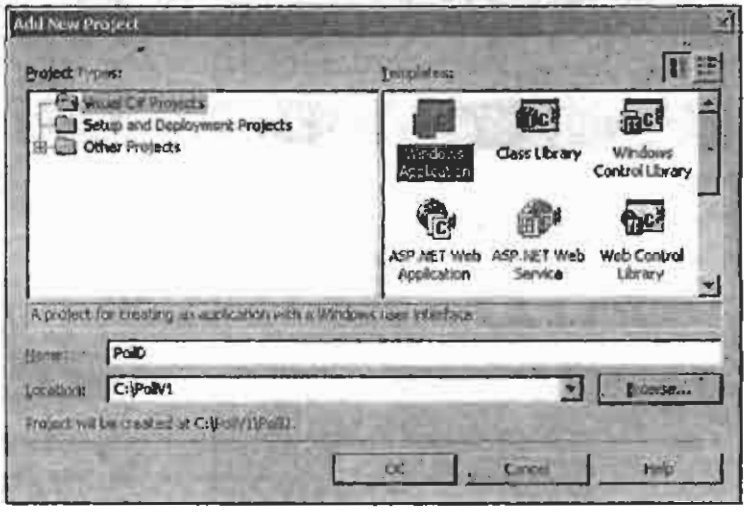
В выходном потоке web-метода `Votes()` мы увидим нечто похожее (см. рис. на стр. 779, верхний), но здесь стоит обратить внимание на то, как просто .NET и web-служба работают с массивом целых значений. Объем работы, выполняемый .NET за кулисами, огромен, однако нам нет никакой необходимости знать о нем что-либо, что позволяет нам выполнять свою работу намного более эффективно.



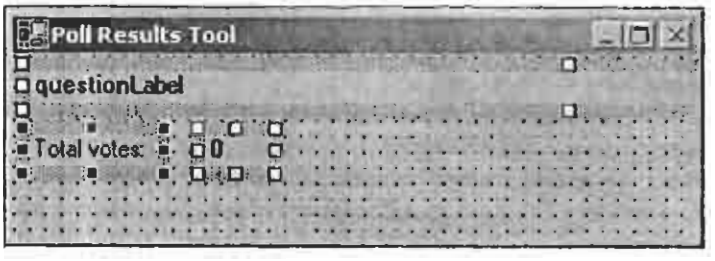
Приложение Windows

Создать web-методы и организовать доступ к ним посредством HTTP оказывается очень легко, но возникает вопрос: каким образом web-методы могут быть использованы? Как можно преобразовать получаемый XML во что-либо полезное? Нам не придется этим заниматься, за нас это все также сделает система .NET.

Мы обладаем функционирующей web-службой, у которой имеются web-методы, обеспечивающие доступ к данным опроса. Теперь переключим внимание на приложение Windows Forms. Мы воспользуемся пунктом Add Project меню File, затем выберем Windows Application и введем имя `PollD` (сокращение от названия приложения Poll Desktop), как показано на рисунке справа.



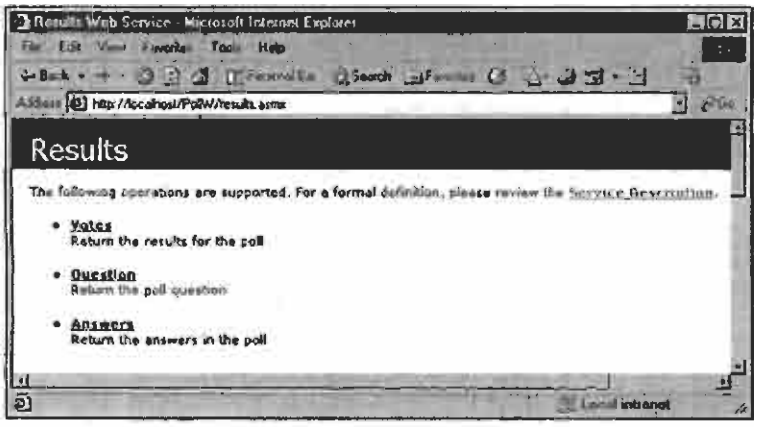
В результате Visual Studio.NET создаст один файл `Form1.cs` с описанием класса и откроет форму в окне разработки, в котором можно непосредственно производить редактирование. Хотя мы планируем создавать большую часть управляющих элементов формы посредством использования кода, три расположенных в самом верху управляющих элемента можно включить, воспользовавшись окном разработки Visual Studio.NET. Первым из них является управляющий элемент `Label` с именем `questionLabel`, который выводит вопрос (см. рис. слева).



Вторым управляющим элементом является также элемент `Label`, который просто выводит текст `"Total votes:"` ("Суммарное количество голосов"). Третий управляющий элемент `Label` называется `totalVotesLabel` и выводит на экран суммарное количество всех голосов, зарегистрированных за время проведения опроса. После добавления названных управляющих элементов форма примет вид, примерно изображенный на рисунке.

Ширина и высота формы задаются позже программным путем. Теперь нужно внести в простую форму реальные данные, воспользовавшись для этих целей web-методом `Question()`, для чего потребуется включить в приложение специальный класс, который называется классом-посредником. В нем содержится код, необходимый для вызова web-метода, кроме того, он упаковывает получаемые в виде XML результаты в форму, которая может быть использована в нашей программе (это осуществляется за счет использования технологии, которая называется SOAP и которая рассматривалась в главе 25).

Создать такой класс-посредник можно двумя способами. Можно либо воспользоваться утилитой `wsdl.exe`, либо использовать для этих целей Visual Studio.NET. Последний способ является более простым, и именно к нему мы прибегнем. Чтобы добавить в проект `PollD` класс-посредник, необходимо щелкнуть правой кнопкой мыши на проекте и выбрать пункт **Add Web Reference**. Открывшееся окно диалога содержит в верхней части поле, в которое вводится строку URL той web-службы, которую мы собираемся использовать. Кроме этого, можно использовать UDDI, являющуюся всемирным хранилищем web-служб, доступ к которым может быть осуществлен через Интернет. Мы не будем использовать UDDI, поэтому просто введем строку URL web-службы в соответствующее поле — `http://localhost/PollW/results.asmx`.



Если теперь нажать клавишу *Return* или щелкнуть мышью на кнопке с зеленой стрелкой, то осуществится переход в окно диалога, где выведена страница web-службы в удобной для чтения форме, какую мы видели раньше. Убедившись в том, что мы обращаемся к правильной web-службе, щелкаем мышью на **Add Reference**, и в этот момент Visual Studio.NET создает класс-посредник (см. рис. слева).

Visual Studio.NET создает в приложении Windows новое пространство имен с именем `PollD.localhost`, в котором будут находится все web-методы данной web-службы. Это пространство имен следует назвать `PollD.PollResults`, щелкнув правой кнопкой мыши на localhost в Solution Explorer, поскольку такое имя более понятно. В пространстве имен `PollD.PollResults` будет использоваться единственный класс — `Results`. В него входят все три метода, которые можно использовать: `Question()`, `Votes()` и `Answers()`.

Вернемся к `Form1.cs`. Пока нет никаких управляющих элементов, позволяющих выводить какие-либо данные опроса. Нам потребуется создавать каждый управляющий элемент в динамике, но прежде всего необходимо получить информацию о том, каково количество ответов в данном опросе, с тем чтобы можно было определить, какое количество управляющих элементов следует создавать и какие размеры должна иметь сама форма. Для простоты в данном примере не используется web-метод `AnswerCount()`, поэтому для определения общего количества возможных ответов мы воспользуемся числом элементов в массиве, который возвращается web-методом `Answers()`. В файл `Form1.cs` нужно включить следующее объявление пространства имен, что даст возможность использовать класс-посредник:

```
using PollD.PollResults;
```

Конструктор в `Form1` сначала содержит строку кода, которая необходима Visual Studio.NET, и именно после нее следует включить код для web-метода `Answers()`:

```
public Form1()
{
    //
    // Необходимо для поддержки Windows Form Designer
    //
    InitializeComponent();
```

Сначала требуется создать экземпляр класса `Results`. Не следует забывать, что это — класс-посредник, благодаря которому можно осуществлять доступ ко всем удаленным web-методам.

```
    Results pollResults = new Results();
```

В следующей строке задается текст, который будет виден на экране, пока форма осуществляет загрузку данных. При нормальной работе этот текст исчезает очень быстро:

```
    questionLabel.Text = "Loading poll data...";
                                        / Загрузка данных опроса... /
```

В следующей строке происходит фактический вызов web-метода `Answers()`. Обратите внимание на то, что это обычный вызов метода — класс-посредник, сгенерированный Visual Studio.NET, скрытно выполняет за нас всю необходимую работу:

```
    string[] pollAnswers = pollResults.Answers();
```

В конструктор класса `Form1` нужно включить переменную с именем `myAnswerCount`, в которой будет храниться счетчик числа ответов, предусмотренных для данного опроса. Поскольку массив, получаемый web-методом `Answers()`, индексируется с нуля и его первый элемент является пустым, то необходимо уменьшить счетчик на единицу:

```
    string[] myAnswerCount = pollAnswers.Lenrth-1;
```

Когда известно количество ответов, можно вычислить размеры формы и включить в нее необходимые управляющие элементы. Каждая строка данных будет включать в себя:

- **Управляющий элемент `Label` для вывода ответа.** Предназначен для вывода текста ответа.
- **Управляющий элемент `Label` для вывода числа голосов.** Предназначен для вывода числа голосов, поданных за этот вариант ответа как в виде числа, так и в процентном отношении к общему числу проголосовавших.
- **Управляющий элемент в виде линейки отображения результатов голосования.** Предназначен для графического представления числа голосов, поданных за этот вариант ответа.

В данной форме будет содержаться не так уж много управляющих элементов, и важно помнить, что они должны использоваться в виде групп. Например, когда мы получаем данные по голосованию, необходим способ, который позволил бы в цикле пройти по всем управляющим элементам для вывода числа голосов, с тем чтобы присвоить свойству `Text` каждого из них соответствующий счетчик голосов.

Чтобы это сделать легко и просто, нужно сформировать массив созданных управляющих элементов `Label`, в котором один массив будет содержать надписи с текстом ответов, второй — надписи с числом голосов и третий — линейки для графического представления результатов голосования.

Следующая строка в `Form1.cs` предназначается для создания массива, в котором содержатся все управляющие элементы `Label`, ответственные за вывод ответов опроса ("C#", "C++" и т. д.). Массив `answerControls` описывается на уровне класса следующим образом:

```
private Label[] answerControls
```

При создании массива используется поле `myAnswerCount`, в котором содержится значение суммарного количества ответов:

```
answerControls = new Label[myAnswerCount];
```

После создания массива можно приступать к организации цикла. При каждом проходе цикла создается управляющий элемент `Label`, задается его местоположение и некоторые другие свойства и ссылка на цикл записывается в массив, что позволит легко обратиться к нему впоследствии. Первым следует оператор `for`, в котором вновь используется поле `myAnswerCount`:

```
for(int i=1;i<= myAnswerCount;i++)
{
```

Для создания нового управляющего элемента Windows Forms Label используется ключевое слово `new`.

```
    Label answerLabel = new Label();
```

Далее создается объект `Location`, который будет использоваться для определения места, где должна появиться данная надпись. Обратите внимание на то, каким образом определяется, что надпись должна располагаться на расстоянии 8 пикселей от левого края и 37 пикселей от верхнего края плюс еще 20 пикселей вниз для каждой последующей надписи (каждая надпись, включая небольшое пустое пространство вокруг нее, имеет высоту около 20 пикселей). Сначала создается самая верхняя надпись, после чего смещение происходит в направлении к нижнему краю:

```
    answerLabel.Location = new System.Drawing.Point(8, 37+(20*i));
```

Создается объект `Size`, который используется для задания размеров надписи — 200 пикселей в ширину и 16 пикселей в высоту. Такие значения оказываются подходящими для оформления страницы в данном случае — вы можете попробовать изменить их и посмотреть, что из этого выйдет:

```
    answerLabel.Size = new System.Drawing.Size(200, 16);
```

В качестве цвета фона задается оттенок белого:

```
    answerLabel.BackColor = Color.AntiqueWhite;
```

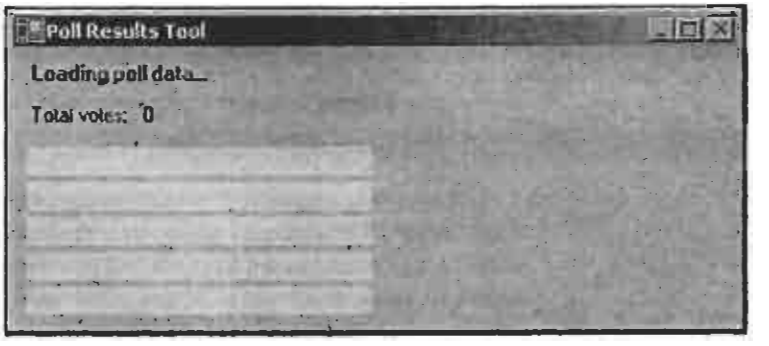
В следующей строке помещается ссылка на управляющий элемент `Label` в массиве. Помните, что в результате этого мы только получаем возможность проходить в цикле по всем ответам, когда возникает необходимость присвоить им соответствующие значения:

```
    answerControls[i-1] = answerLabel;
```

Наконец, используется метод `Add()` свойства `Controls` формы для включения надписи в форму:

```
        Controls.Add(answerLabel);
    }
```

Если теперь построить проект и запустить приложение, то можно увидеть множество надписей, которые генерируются автоматически. Обратите внимание на то, что приложение связалось с web-сервером и запросило у него значение общего числа ответов. И хотя в данном примере приложение Windows (`PollD`) и web-служба (внутри `PollW`) выполняются на одной и той же машине, они могут быть разнесены по разным машинам путем внесения весьма незначительных изменений.

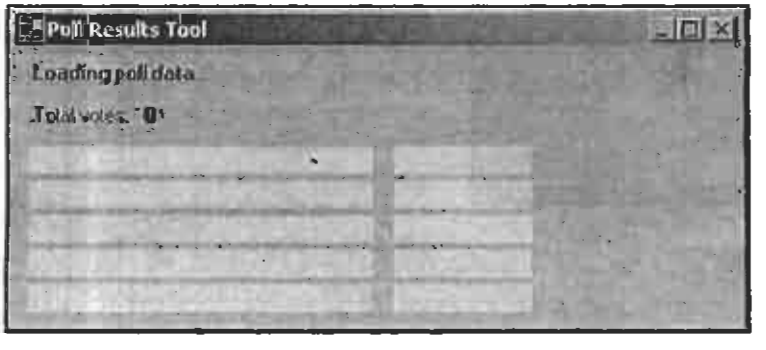


Теперь включим управляющие элементы, которые отвечают за вывод числа голосов в более наглядной графической форме. Приведенный ниже код — это следующая часть конструктора `Form1.cs`. Этот код практически идентичен рассмотреному, за исключением того, что надписи сдвигаются еще дальше вправо (на 220 пикселей вместо 8), а ссылки на них помещаются в массив, называющийся `voteControls` и описываемый на уровне класса следующим образом:

```
    private Label[]voteControls;
```

Вернемся к коду конструктора и добавим еще несколько управляющих элементов:

```
    voteControls = new Label[myAnswerCount];
    for(int i=1;i<= myAnswerCount;i++)
    {
        Label voteLabel = new Label();
        votesLabel.Location = new System.Drawing.Point(220, 37+(20*i));
        votesLabel.Size = new System.Drawing.Size(200, 16);
        votesLabel.BackColor = Color.AntiqueWhite;
        voteControls[i-1] = votesLabel;
        Controls.Add(votesLabel);
    }
```



Добавив этот код, мы сможем увидеть, что теперь в форме присутствуют два столбца управляющих элементов (см. рис. слева).

Для вывода графических индикаторов голосования воспользуемся управляющим элементом `ProgressBar`. Начать следует как обычно с создания массива, который впоследствии будет использоваться для присваивания в цикле значений этим управляющим элементам. Переменная `voteBarControls` описывается на уровне класса таким образом:

```
    private ProgressBar[] voteBarControls.
```

В конструктор формы необходимо добавить следующее:

```
voteBarControls = new ProgressBar[myAnswerCount];
```

Далее мы проходим один раз в цикле все ответы — точно так же, как для предшествующих управляющих элементов Label:

```
for(int i=1;i<= myAnswerCount;i++)
{
```

Затем для формы создается управляющий элемент ProgressBar:

```
ProgressBar progressBar = new ProgressBar();
```

После этого создается объект Point, который используется для задания местоположения управляющих элементов ProgressBar в форме. Каждая линейка графического представления рисуется в 310 пикселях от левого края и в 37 пикселях от верхнего края плюс еще 20 пикселей вниз для каждой очередной линейки. Они оказываются выведенными вровень с двумя определенными ранее столбцами управляющих элементов Label:

```
ProgressBar.Location = new System.Drawing.Point(310, 37+(20*i));
```

В качестве минимального значения для линейки графического представления задается значение, равное нулю. Позднее, в качестве максимального значения мы зададим наибольшее количество голосов, отданное за какой-либо из ответов:

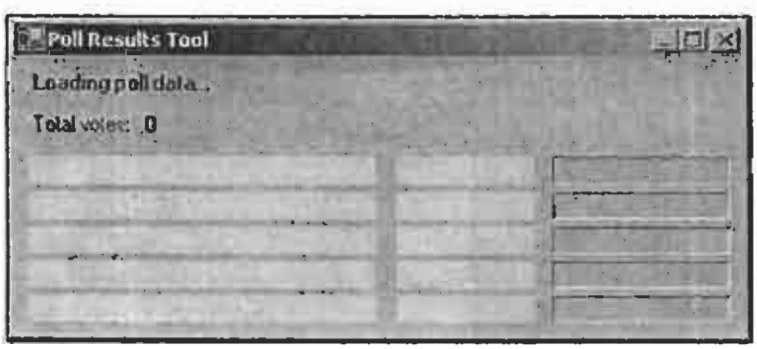
```
ProgressBar.Minimum = 0;
```

Далее определяются ширина линейки графического представления, равная 103 пикселям, и ее высота, равную 16 пикселям:

```
ProgressBar.Size = new System.Drawing.Size(103, 16);
```

В завершение ссылка на управляющий элемент заносится в массив линеек графического представления, а сама линейка добавляется в форму:

```
    voteBarControls[i-1] = progressBar;
    Controls.Add(progressBar);
}
```



Если запустить приложение Windows, то можно увидеть, что теперь есть три столбца управляющих элементов. Первые два — это надписи, а в третьем находятся линейки графического представления. Обратите внимание на возможность доступа к каждому столбцу посредством созданных массивов; это существенно упрощает присваивание данных управляющим элементам (см. рис. слева).

Чтобы быть уверенными в том, что форма всегда будет иметь надлежащие размеры с учетом количества содержащихся в ней управляющих элементов, нужно добавить следующие две строки кода, определяющие высоту и ширину формы:

```
this.Height = 87 + (20*myAnswerCount);
this.Width = 428;
```

После создания формы встает задача наполнить ее конкретными данными, при этом требуется постоянно обновлять ее содержимое. С этой целью нужно создать специальный инструмент, который будет запрашивать данные с web-сайта каждые несколько секунд. В состав .NET включен объект с именем `Timer` (таймер), которым мы и воспользуемся. При использовании объекта `Timer` требуется задать интервал времени и вызываемый метод. Каждый раз в момент истечения интервала времени будет вызываться указанный метод, поэтому все, что требуется,— это написать такой метод и запустить таймер. Метод, который мы собираемся создавать для получения данных от web-службы и переноса их в управляющие элементы формы, называется `GetPollData()`.

Во-первых, необходимо включить следующий оператор в начало кода:

```
using System.Timers;
```

Затем требуется создать экземпляр объекта `Timer` с помощью ключевого слова `new`:

```
System.Timers.Timer aTimer = new System.Timers.Timer();
```

Затем нужно включить в него обработчик событий, который определяет, какой именно метод должен быть вызван в момент истечения интервала времени. Именно здесь определяется, что в этом случае должен вызываться метод `GetPollData()`:

```
aTimer.Elapsed += new ElapsedEventHandler(GetPollData);
```

После этого задается значение временного интервала и запускается таймер. В данном конкретном случае устанавливается интервал в 5 сек., хотя в реальной ситуации это значение должно быть настраиваемым:

```
aTimer.Interval=5000;
aTimer.Enabled=true;
```

Теперь есть набор управляющих элементов в форме и таймер, который будет вызывать метод `GetPollData()`, записывающий данные в форму каждые пять секунд. Первое, что должен выполнять описанный в `Form1` метод `GetPollData()`, это создавать экземпляр объекта класса-посредника web-службы, что позволит обращаться к созданным web-методам:

```
public void GetPollData( object source, ElapsedEventArgs e )
{
   Results pollResults = new Results();
```

Чтобы задать в форме текст вопроса, нужно осуществить вызов web-метода `Question` через объект-посредник (`pollResults`) и использовать его для присваивания значения свойству `Text` управляющего элемента `questionLabel`:

```
questionLabel.Text = pollResults.Question();
```

Следующие две строки кода получают массив ответов и присваивают переменной `myAnswerCount` общее число ответов для данного опроса:

```
string[] pollAnswers = pollResults.Answers();
int myAnswerCount = pollAnswers.Length-1;
```

Теперь необходимо вычислить два значения, которые не предоставляются данными web-методами. Первое из них — это суммарное количество голосов. Второе — это максимальное количество голосов, поданных за один из ответов (это значение будет использовано для определения верхней границы линеек графического представления). Для определения этих двух значений следует воспользоваться web-методом `Votes()`, который возвращает массив счетчиков голосов, а затем пройти

по этому массиву в цикле, обновляя текущие значения переменных `totalVotes` (общее количество голосов) и `highestVote` (максимальное количество голосов) по мере просмотра этого массива:

```
int totalVotes = 0;
int highestVote = 0;
int[] pollVotes = pollResults.Votes();
for(int i=1;i<=myAnswerCount;i++)
{
   totalVotes = totalVotes + pollVotes[i];
   if pollVotes[i] > highestVote)
      highestVote = pollVotes[i];
}
```

Для вычисления процентного отношения голосов потребуется целая переменная:

```
int percentageVote;
```

В нескольких последующих строках кода мы будем проходить в цикле по всем ответам и присваивать соответствующие значения в каждый из трех столбцов управляющих элементов формы — сначала текст ответа, затем результаты голосования (в том числе и в виде процентов) и, наконец, линейку графического представления. Для этого мы организовываем цикл `for`:

```
for(int i=1;i<=myCountAnswer;i++)
{
```

Мы присваиваем текст ответа в первый столбец управляющих элементов формы, для чего используется свойство `Text` управляющего элемента `Label`. Доступ к соответствующему управляющему элементу `Label` осуществляется с помощью массива управляющих элементов, созданного ранее. Обратите внимание на то, что, поскольку индекс массива управляющих элементов в отличие от индекса массива ответов отсчитывается с нуля, мы вычитаем единицу из его значения:

```
answerControls[i-1].Text = pollAnswers[i];
```

Для присваивания значений полям, в которых содержится число голосов, используется аналогичная процедура, однако сначала необходимо определить процентное соотношение. Для этого сначала следует разделить число голосов для данного ответа на суммарное число голосов. Результат деления записывается в переменную типа `float`:

```
float voteFraction = ((float)pollVotes[i]/(float)totalVotes);
```

Чтобы получить значение, выраженное в процентах, нужно умножить полученное число на сто:

```
percentageVote = (int)(100 * voteFraction);
```

Чтобы иметь возможность вывести счетчик голосов, нужно присвоить копию числа голосов в текстовом виде свойству `Text`, соответствующему управляющему элементу, который хранится в массиве управляющих элементов для счетчиков голосов:

```
voteControls[i-1].Text = pollVotes[i].ToString()+" ";
```

В конец строки нужно добавить текстовое представление процентного соотношения, которое для наглядности заключается в круглые скобки и после которого добавляется символ '%':

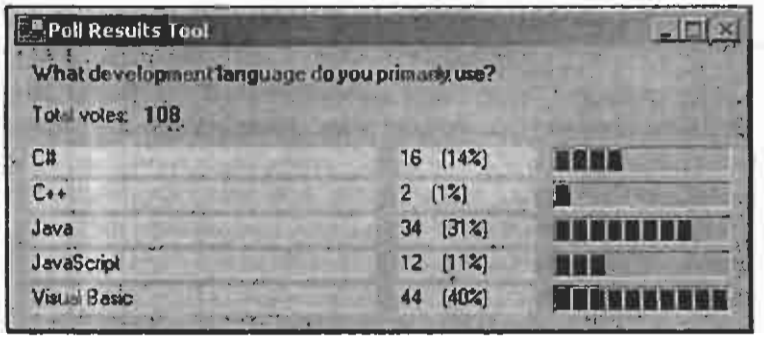
```
voteControls[i-1].Text += "(" + percentageVote.ToString() + "%)";
```

Для задания линейки графического представления необходимо устаиовить максимальное значение, которого она может достигать, и фактическое значение. Для задания максимального значения используется ранее определенная переменная `highestVote`, а затем задается фактическое значение счетчика голосов с помощью свойства `Value`:

```
            voteBarControls[i-1].Maximum = highestVote;
            voteBarControls[i-1].Value = pollVotes[i];
        }
```

В завершение выводится суммарное количество голосов путем присвоения свойства `Text` управляющему элементу `totalVotesLabel`:

```
        totalVotesLabel.Text = totalVotes.ToString();
    }
```



Если теперь построить проект и запустить приложение, то перед нами сначала предстанет незаполнеиная форма, однако очень скоро будет запущен объект `Timer`, и все поля будут наполнены соответствующими значениями, полученными с web-сервера (см. рис. слева).

Если же запустить консольное приложение или проголосовать, воспользовавшись для этого web-приложением, то можно увидеть, что не позднее, чем через пять секунд инструмент для вывода результатов опроса правильно отобразит изменившнеся значения.

Один человек — один голос

Если ставится задача ограничить число раз, которое один пользователь может проголосовать, то для ее решения необходимо иметь возможность устанавлнвать факт, что данный пользователь уже голосовал ранее. Поскольку основным используемым нами для регистрацин голосования интерфейсом является web-страница, то нмеется преимущество использования куки, которые представляют собой идеальное средство для хранения информации, подобиой этой. Куки — это небольшой блок информации, который сохраняется браузером посетителя web-сайта на его жестком диске по запросу web-сайта, и каждый раз, когда посетитель возвращается на данный сайт, куки посылается обратно. Если в полученном куки будет отражен факт участия в голосовании, то можно в будущем игнорировать любые попытки даиного посетителя проголосовать еще раз. Главным недостатком такого подхода является то, что если использование куки будет заблокировано, то любой посетитель сможет голосовать произвольное чнсло раз.

Первое, что требуется сделать,— это внести небольшие изменения в файл `default.aspx` в проекте web-приложения (`PollW`), которые позволят просматривать куки и наблюдать за тем, что делает наша программа. Изменение, отмеченное в программе (`Trace="true"`), означает, что когда открывается файл `default.aspx` в браузере, то видны несколько страниц с отладочной информацией, в том числе подробная информация о куки, полученных страницей. Сейчас те из вас, кто ранее работал с ASP 3.0, возможно, будут очень снльно удивлены:

```
<%@ Page language="c#" Trace="true" ... Inherits="PollW.WebForm1" %>
```



Если теперь открыть в браузере `default.aspx`, то можно обнаружить всего один куки, который используется ASP.NET для хранения информации о сеансе (см. рис. слева).

Теперь есть возможность создать некий код, позволяющий сохранить куки, и посмотреть, что будет происходить при его выполнении. Обратимся к классу `Poll`. Мы будем сохранять куки на компьютере посетителя, после того как он проголосует,— в конце метода `Vote()`. В конце этого метода мы записывали данные опроса обратно в XML-файл и закрывали поток:

```
myDataSet.WriteXml(myStream,XmlWriteMode.IgnoreSchema);
myStream.Close();
```

Чтобы сохранить куки, следует включить сюда дополнительный код. Во-первых, нам необходимо выполнить проверку на предмет того, выполняется ли данный метод в рамках `HttpContext`. Это проверка позволит гарантировать, что код для работы с куки будет выполняться только в том случае, если метод вызывается из страницы ASP.NET, и будет игнорироваться во всех остальных случаях, например, при вызове из консольного приложения или приложения Windows Forms.

Чтобы иметь возможность обратиться к классу `HttpContext`, необходимо включить в программу ссылку на пространство имен `System.Web`:

```
using System.Web;
```

Теперь нужно вставить код после записи в XML-поток:

```
if (HttpContext.Current != null)
{
```

Для создания куки потребуется создать экземпляр объекта с помощью класса `HttpCookie`. У каждого куки имеются свое собственные имя и значение. Имя всегда будет одним и тем же (`WroxPoll`), а вот значение будет уникальным для каждого конкретного запроса, поскольку необходимо гарантировать, что при проведении нового опроса любой пользователь сможет успешно зарегистрировать свой голос:

```
    HttpCookie myCookie = new HttpCookie("WroxPoll");
```

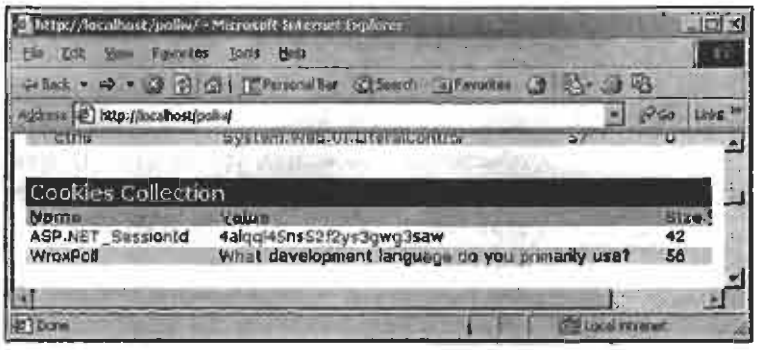
В качестве значения, уникального для данного опроса, используется задаваемый в нем вопрос:

```
    myCookie.Value = (myQuesqtion);
```

Для получения текущего HTTP-ответа используется объект `HttpResponse`, и следует обратиться к его свойству Response, которое представляет собой данные, отправляемые обратно браузеру пользователя. Метод `Add()` свойства `Cookies` используется для задания того, что новый куки должен быть добавлен в HTTP-поток, с тем чтобы браузер пользователя записал его на диск:

```
    HttpContext.Current.Response.Cookies.Add(myCookie);
}
```

Если заново построить проект, открыть страницу `default.aspx`, проголосовать, а затем повторно загрузить `default.aspx`, то часть трассировочного вывода, которая посвящена куки, будет содержать именно наш куки (см. рис. справа).

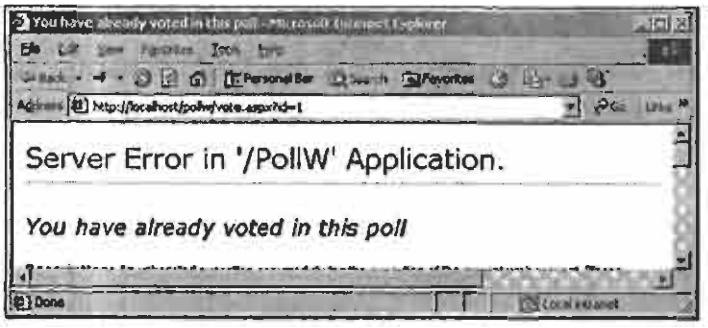


После успешного завершения процесса сохранения информации в куки на машине пользователя появится возможность определять, принимал ли уже данный пользователь участие в текущем опросе. Чтобы заблокировать возможность повторного голосования, нужно включить в начало метода `Vote()` следующий код:

```
if (HttpContext.Current.Request.Cookies["WroxPoll"] != null)
{
   if (HttpContext.Current.Request.Cookies["WroxPoll"].Value == myQuestion
   {
      throw (new Exception("You have already voted in this poll"));
                                  / Вы уже принимали участие в данном опросе /
   }
}
```

Сначала выполняется проверка на предмет того, существует ли вообще данный куки, и, убедившись, что он не равен `null`,— проверка на предмет того, не совпадает его значение с вопросом активного в настоящий момент опроса.

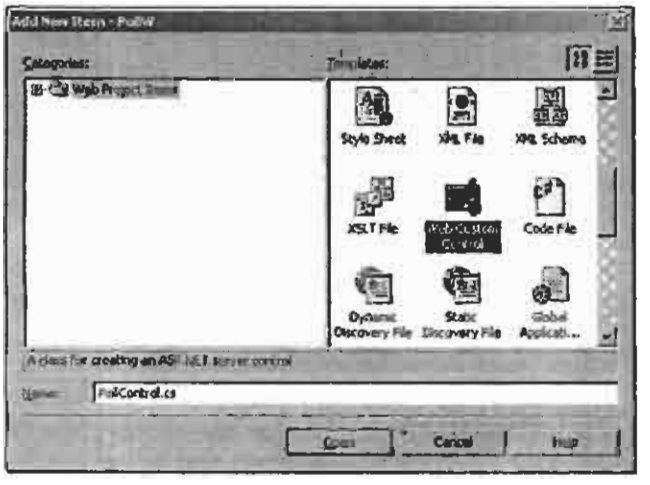
В приложении для коммерческого использования вы, очевидно, захотите представить отказ в голосовании в более дружественной по отношению к пользователю форме, чем та, которая используется в данном случае, но для целей нашего примера окажется вполне достаточно, если наличие куки, значение которого совпадает с текстом вопроса текущего опроса, будет приводить к возникновению исключительной ситуации. Помните, что перед использованием `HttpContext` следует осуществить проверку на предмет того, не равен ли он `null`, чтобы случайно не обратиться к куки другого типа приложения, отличного от приложения ASP.NET (см. рис. слева).



Выясняется, что если вы зайдете в XML-файл (`C:\polls.xml`) и измените вопрос опроса, то вы снова получите возможность принимать участие в голосовании. Это объясняется тем, что наша проверка на предмет того, принимал ли пользователь участие в голосовании, основана на вопросе опроса. Стоит изменить вопрос, как значение куки и текст вопроса перестанут совпадать, а голосование будет зарегистрировано. Если голосование регистрируется, то приложение Windows Forms выводит новое значение суммарного количество голосов после его обновления (т. е. не позднее пяти секунд).

Управление опросами

Одно из преимуществ .NET перед предшествующими технологиями заключается в том, что возможность повторного использования создаваемого программного обеспечения несколько упрощается. В этом разделе мы собираемся создать пакет web-сайта для проведения опросов, с тем чтобы иметь возможность вставлять этот сайт в любую страницу ASP.NET при помощи всего лишь пары строк кода. На данном этапе необходимо выполнить довольно значительный объем работы, для того чтобы интегрировать процедуру проведения опроса в какую-либо другую web-страницу, поскольку для этого потребуется скопировать соответствующую информацию как из файла с расширением `.aspx`, так и из файла "с кодом поддержки", который имеет расширение `.aspx.cs`. Мы будем создавать управляющий элемент таким образом, что в качестве дополнительной возможности он будет выводить страницу с опросом, и если данный пользователь уже проголосовал ранее, он сможет только ознакомиться с результатами опроса — страница голосования выводиться не будет.



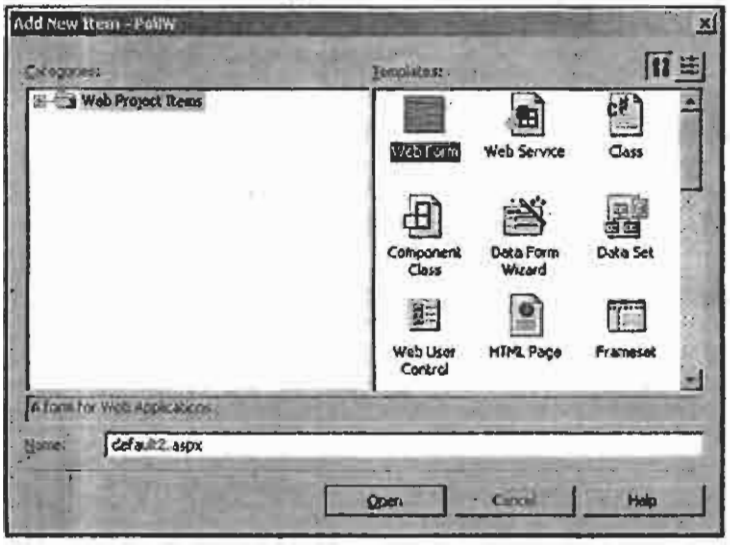
Для того чтобы создать пакет для проведения голосования, нужно разработать нестандартный управляющий элемент, который будет реализован в классе, наследуемом от класса `Control`, описанного в .NET Framework. Для начала в Visual Studio.NET следует щелкнуть правой кнопкой мыши на проекте web-приложения (`PollW`), выбрать меню **Add**, а в нем пункт **Add New Item**. В появившемся диалоговом окне нужно выбрать иконку **Web Custom Control** и ввести в поле **Name** `PollControl.cs`, как это проиллюстрировано на рисунке слева.

Хотя Visual Studio.NET генерирует несколько строк кода, для того чтобы нам было с чего начинать (при этом реализуется свойство `Text` создаваемого управляющего элемента), однако нам все это не понадобится, поэтому для начала работы нужно привести класс к еще более простому виду:

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.ComponentModel;

namespace PollW
{
   /// <summary>
   /// Краткое описание класса PollControl.
   /// </summary>
   public class PollControl : WebControl
   {
      protected override void Render(HtmlTextWriter output)
      {
         output.Write("This is my Poll Control");
                        // Это управляющий элемент для проведения опросов //
      }
   }
}
```

Заметьте, что все, что на самом деле необходимо выполнить,— это создать класс, который наследуется от `System.Web.UI.WebControls.WebControl`. Сделав это, мы реализуем переопределение метода `Render()`, в котором генерируем HTML, который мы планируем выводить. Существует также возможность написать такой управляющий элемент, который будет создавать себя сам посредством объединения других управляющих элементов, но в настоящем примере мы будем генерировать HTML.



Попытаемся вставить в страницу простой нестандартный управляющий элемент и посмотреть, что для этого придется сделать. Во-первых, добавим к нашему проекту новую страницу, которую назовем `default2.aspx` и которую мы будем использовать в качестве контейнера нестандартного управляющего элемента. Это достигается за счет использования пуикта Add Web Form из меню Project, как показано на рисунке слева.

Мы приведем в порядок содержимое `default2.aspx`, чтобы ничего не усложнять и четко представлять себе, что происходит:

```
<%@ Page language="c#" Codebehind= default2.aspx.cs" ..... %>
<html>
   <head>
   </head>
   <body>
   </body>
</html>
```

Для того чтобы вставить управляющий элемент для проведения опросов в страницу, следует добавить всего две строки:

```
<%@ Page language="c#" Codebehind= default2.aspx.cs" ..... %>
<%@ Register TagPrefix="Wrox" Namespace="PollW" Assembly="PollW" %>
<html>
   <head>
   </head>
   <body>
      <Wrox:PollControll runat="server"></Wrox:PollControl>
   </body>
</html>
```

В первой из двух добавленных строк объявляется префикс тегов `TagPrefix` с именем Wrox. Это означает, что если ASP.NET встречает на странице любой серверный управляющий элемент, который начинается с `<Wrox:`, то система начинает осуществлять поиск в пространстве имен `PollW` в модуле `PollW` управляющего элемента с таким же именем. В данном случае мы указали `<Wrox:PollControl>`, поэтому ASP.NET будет осуществлять поиск в пространстве имен `PollW` класса `PollControl`. Вторая добавленная строка представляет собой серверный управляющий элемент, и именно в ней мы фактически вставляем опрос в страницу. Обратите

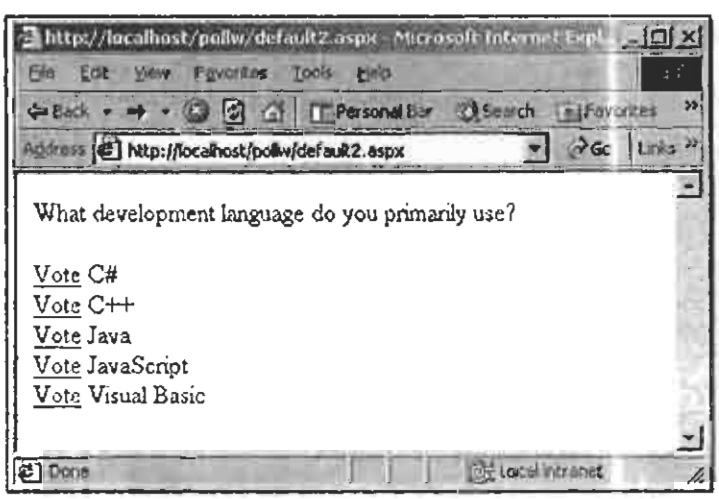


вниманне на то, что мы включили в нее атрибут `runat`. Если же теперь открыть файл `default2.aspx` в своем браузере, то можно увидеть текст, выводящийся методом `PollControl Render()` (см. рис. слева).

Мы успешно вставили в страницу нестандартный управляющнй элемент; теперь добавим в метод `Render()`, находящийся в файле `PollControl.cs`, код, который позволит выводить опрос на экран. Упомянутый код оказывается практически идентичным коду для "файла поддержки" `default.aspx.cs`. Не забудьте, что в начале файла необходимо поместить оператор `using Polling`:

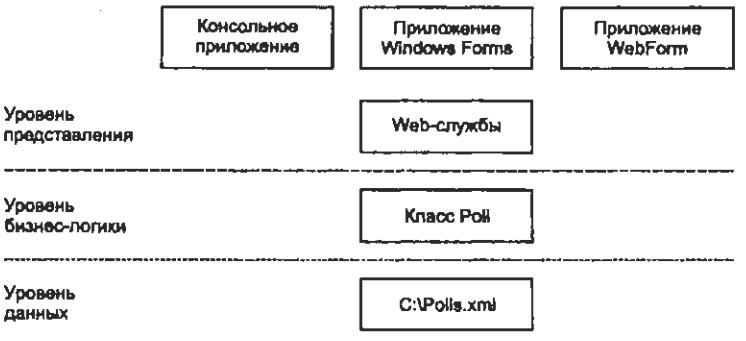
```
protected override void Render(HtmlTextWriter output)
{
   string Html = " ";
   Poll aPoll = Poll.Current();
   Html += aPoll.Question + "<br /><br />";
   for(int i=1;i<=aPoll.AnswerCount;i++)
   {
      HtmlLink += "<a href=\"vote.aspx?id="+i+"\">Vote</a>";
      HtmlLink += " " + aPollAnswers[i] + "<br />";
   }
   output.Write(Html);
}
```

Если теперь построить проект заново и открыть файл `default2.aspx`, то можно увидеть то же самое, что и в случае `default.aspx` за исключением того, что эту версию гораздо проще использовать повторно. Все, что необходимо сделать для того чтобы использовать ее на некотором сайте, — это сначала убедиться в доступности файла `vote.aspx`, затем убедиться в доступности модулей, содержащих классы `Poll` и `PollControl`, после чего просто добавить две строки кода к странице ASP.NET. В результате этих действий получилась некая оболочка, объемлющая отдельные функциональные возможности. Мы назвали ее управляющим web-элементом. Этот подход позволяет объединять различные функциональные возможности и облегчает их повторное использование. Результат можно видеть на рисунке справа.



Трехзвенность и приложение Poll

Нами создано приложение, которое может функционировать и как консольное приложение, и как web-служба, и как приложение Windows Forms, и как Интернет-интерфейс ASP.NET. Для того чтобы наглядно представлять себе, как все эти отдельные части соотносятся друг с другом, слева приведена диаграмма, на которой изображены основные элементы приложения `Poll` и проиллюстрировано, каким образом это приложение вписывается в модель трехзвенной архитектуры, состоящую из уровня данных, уровня бизнес-логики и уровня представления.



Заметьте, что несмотря на то, что приложение Windows Forms обычно взаимодействует с классом `Poll` посредством web-службы, ему не придется этого делать, если оно будет запущено на той же машине, что и класс `Poll` (другими словами, на web-сервере).

Итоги

Данный практический пример позволил продемонстрировать, каким образом можно применить теорию по C# и .NET, изучавшуюся нами на протяжении всей настоящей книги, в ситуации из реальной жизни, где перед нами возникла задача создать приложение, которое было бы доступно посредством различных пользовательских интерфейсов. В этом практическом примере воссоздается ситуация из реальной жизни в том смысле, что мы на первой итерации создаем первоначальную версию приложения, а затем добавляем в него новые функциональные возможности. В мире коммерческих продуктов программное обеспечение не пишется — оно модифицируется, и именно это мы и попытались здесь продемонстрировать.

В качестве краткого напоминания перечислим приемы и технологии, которые были использованы в этом практическом примере:

- Бизнес-объекты (класс `Poll`)
- Генерирование исключительных ситуаций
- Нестандартный управляющий элемент ASP.NET
- Серверные управляющие элементы ASP.NET
- Статически методы класса
- Наборы данных XML и ADO.NET
- Web-службы и web-методы
- `System.Timer` (таймер)
- HTTP-куки
- Класс `HttpContext`

- Уровень представления, поддерживающий множество интерфейсов
- Консольное приложение
- Приложение Windows Forms
- Трехзвенная архитектура
- Объектно-ориентированная разработка
- Тестирование программного обеспечения

Надеемся, что вы получили удовольствие от чтения этой главы, а информация, которую вы из нее почерпнули, вдохновит вас на создание своего собственного приложения на C#. .NET в сочетании с такими методами, как трехзвенная архитектура и объектно-ориентированная разработка, оказывается чрезвычайно мощной платформой для разработки программных приложений, а C# — наиболее эффективным из всех языков программирования, входящих в состав .NET.



Практический пример 2 — Менеджер рассылки новостей в режиме онлайн

Программа рассылки новостей

В предшествующих главах настоящей книги вы познакомились с тем, как программировать на языке C# и как использовать этот язык программирования в приложениях Windows и простых приложениях ASP.NET. Теперь мы собираемся продемонстрировать, каким образом можно использовать многое из того, с чем вы познакомились, для создания web-приложения, которое может быть использовано в реальном web-сайте .NET.

Ниже перечислены некоторые из тем, с которыми вы познакомитесь или которые вы сможете лучше усвоить в результате прочтения этого практического примера:

- Создание законченных web-приложений с помощью ASP.NET
- Использование некоторых серверных управляющих элементов ASP.NET для создания пользовательского интерфейса с богатыми возможностями
- Использование кода поддержки для разделения основного кода от пользовательского интерфейса
- Создание файла web-конфигурации для хранения установок пользователя для данного приложения и для обеспечения безопасности базового уровня
- Использование объектно-ориентированных возможностей C# для создания и применения классов, в которых инкапсулирована бизнес-логика приложения
- Использование некоторых классов .NET Framework для работы с файлами, почтой и базами данных
- Запись в базу данных и чтение из базы данных с помощью управляемого провайдера OLE DB и классов ADO.NET
- Распространение компилированных приложений ASP.NET и их запуск через IIS

Общий обзор приложения

Любому web-мастеру известны ключевые элементы, необходимые для разработки успешного web-сайта. Вот некоторые из них:

- Хорошее наполнение (контент)
- Организация (удобство поиска и навигации)
- Привлекательный дизайн
- Взаимодействие с пользователем

Успешный web-сайт должен обладать свежим, регулярно обновляемым и интересным содержимым. Список адресов электронной почты может оказаться весьма полезным для постоянного уведомления подписчиков о новых возможностях — в качестве напоминания посетителям и стимула для регулярного посещения сайта.

Управление таким списком может представлять определенные трудности. Администраторам приходится сопровождать список подписчиков, иметь дело с заявками на подписку и с отказами от подписки, а также отслеживать все отправленные сообщения.

Приложение, представляемое в настоящей главе, сможет оказать помощь в автоматизации управления одним из таких списков. Эта программа — Менеджер рассылки новостей в режиме онлайн — позволяет автоматизировать большую часть задач, связанных с подпиской и управлением рассылкой новостей, а также упростить выполнение администратором остальных действий. Эта программа позволяет:

- Управлять множественными списками рассылок, давая администратору возможность добавлять, исключать или вносить изменения в списки и их свойства.
- Осуществлять управление подписчиками из всех списков с возможностью задавать и изменять имя подписчика и относящуюся к нему персональную информацию.
- Рассылать подписчикам сообщения по электронной почте в виде обычного текста или в HTML-формате. Все электронные сообщения могут автоматически снабжаться именем и адресом подписчика, содержать ссылку, позволяющую произвести отказ от подписки, а также любую другую информацию о списке (например, его название и описание).
- Управлять архивом новостей, рассылавшихся по всем спискам ранее, предоставляя администратору возможность читать и копировать предыдущие сообщения.
- Создавать HTML-форму, позволяющую получать данные пользователя без необходимости писать какой бы то ни было код. Эта функциональная возможность оформлена в виде программы-мастера — администратор выбирает список, для которого ему требуется создать форму, задает несколько опций, после чего мастер самостоятельно создает весь необходимый код.
- Задавать некоторые опции, например, имя и адрес отправителя, подпись, которая будет автоматически добавляться к каждому сообщению, предмет сообщения по умолчанию, а также сообщения о подписке и об отказе от подписки.

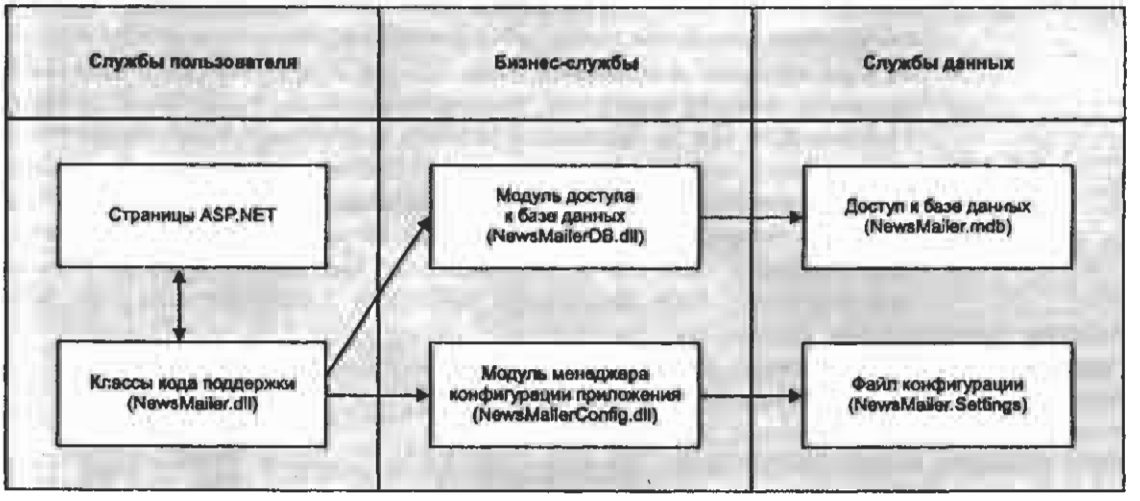
- Использовать возможность изменения внешнего представления приложения (цвета и размеры шрифтов) без внесения изменения в HTML-код или загрузки новых файлов.

Самым существенным преимуществом такого решения является то, что больше не придется вручную управлять группой пользователей: приложение добавляет их в базу данных и исключает их оттуда. Вторым, и также важным преимуществом является то, что администратор получает возможность следить за подписчиками web-сайта в режиме онлайн, общаться с ними и даже вносить изменения во внешнее представление web-сайта из любой точки планеты. Все, что ему для этого требуется,— это компьютер с web-браузером и выход в Интернет.

В данном примере предполагается, что для разработки приложения вы будете использовать Visual Studio.NET. Если это не так, то в программе, которую вы можете загрузить, содержатся указания как откомпилировать это приложение из командной строки.

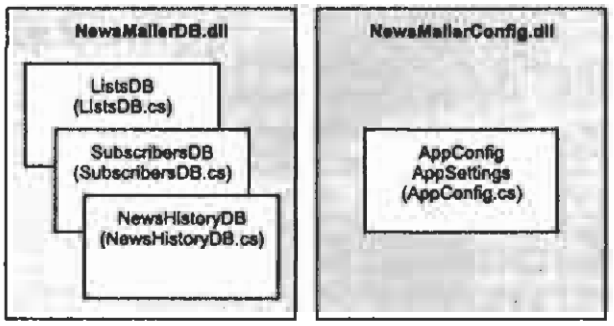
Модель приложения

Хорошие приложения, построенные на основе использования баз данных, обычно разбиваются на уровни для представления, бизнес-логики и данных. Такая модель упрощает сопровождение приложения, а также дает лучшую масштабируемость, поскольку допускает установку приложения на различных серверах. Даже если приложение, которое мы собираемся разрабатывать, не требует повышенной масштабируемости, все равно это тот путь, по которому следует двигаться. Графическое представление структуры нашего приложения представлено на следующей диаграмме:



Уровень представления (службы пользователя) состоит из страниц ASP.NET с их кодом поддержки (откомпилированным в модуль). Уровень бизнес-служб содержит в себе еще два модуля, которые обеспечивают простой доступ к данным. Настоящий раздел посвящен подробному объяснению реализации каждого из этих уровней и модулей.

Объекты данных и бизнес-объекты

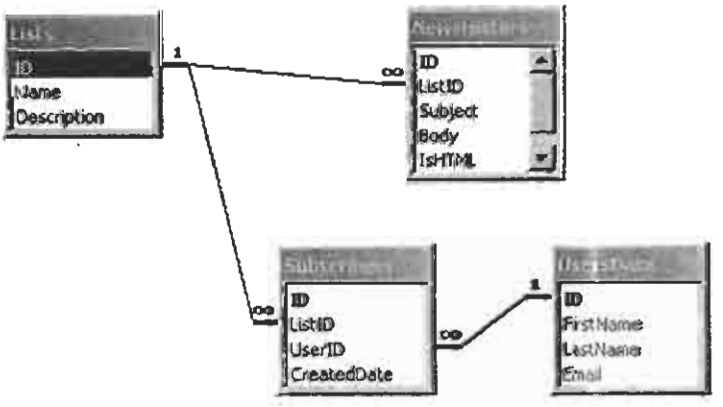


Мы начнем обсуждение с уровня данных и бизнес-уровня. На диаграмме, приведенной на рисунке слева, показаны два модуля бизнес-уровня и общие классы, входящие в их состав.

Модуль `NewsMailerDB.dll` обеспечивает доступ к базе данных, в нем хранятся названия списков рассылки, подписчики и предыдущие сообщения. Модуль `NewsMailerConfig.dll` обеспечивает доступ к XML-файлу, в котором хранятся установки и где все опции приложения могут считываться и изменяться.

Модель базы данных

В целом приложение основывается на использовании базы данных, поэтому начнем обсуждение проекта с выбора и обсуждения модели базы данных. Мы решили использовать базу данных Access вместо SQL Server по целому ряду причин. Базу данных Access проще конфигурировать, кроме того, ее наличие предусматривается в большинстве планов по работе с web-серверами, в то время как наличие SQL Server предусматривается только в наиболее сложных (и более дорогих) проектах по использованию web-серверов. Кроме того, поскольку выигрыш от использования менеджера рассылки новостей прежде всего получают небольшие web-сайты, мы предпочли написать приложение, которое может быть использовано всеми. Более того, использование SQL Server оправдано для широкомасштабных сайтов, при обращении к которым многим пользователям одновременно требуется установление соединения с базой данных. В нашем приложении регулярно устанавливать соединение с базой данных будет только администратор. Пользователям потребуется соединение только для подписки или отказа от подписки, однако такого рода операции выполняются очень быстро, и представляется маловероятным, что очень большое количество пользователей решит подписаться одновременно. Поэтому если вам не приходится работать с десятками тысяч подписчиков, база данных Access является более адекватной. Тем не менее, по ходу обсуждения будет сделано несколько кратких замечаний, как можно перенести базу данных на SQL Server, чтобы воспользоваться ее мощью и скоростью. Совершенно очевидно, что если некоторые части сайта уже используют базу данных SQL Server (например, при разработке сайта электронной коммерции и использовании SQL Server для хранения наименований продукции и информации о ней), вам следует интегрировать в нее и менеджера рассылки новостей, и все таблицы, с тем чтобы избежать использования двух разных баз данных.



Перейдем к обсуждению модели базы данных. БД состоит из четырех таблиц, которые связаны между собой отношениями, изображениями на рисунке справа.

Теперь мы переходим к подробному описанию этих таблиц и их отношений.

Таблица Lists

Таблица Lists используется для хранения информации обо всех доступных списках рассылки и имеет следующую структуру:

Название столбца	Тип	Размер	Допускается значение null	Описание
ID	AutoNumber		Нет	Уникальный идентификатор списка рассылки
Name	Text	50	Нет	Имя списка рассылки
Description	Text	250	Да	Необязательное описание списка рассылки

Таблица UsersData

В таблице UsersData хранится персональная информация пользователей. Таблица имеет следующую структуру:

Название столбца	Тип	Размер	Допускается значение null	Описание
ID	AutoNumber		Нет	Уникальный идентификатор пользователя
FirstName	Text	50	Нет	Имя пользователя
LastName	Text	50	Нет	Фамилия пользователя
Email	Text	50	Нет	Адрес электроиной почты пользователя

В даиной таблице не содержится указания на то, к какому списку рассылки приписан пользователь, поскольку одии пользователь может подписаться сразу на несколько различных списков. Наилучшая практика — избегать повторения одних и тех же деталей. К этому мы вынуждены были бы прибегнуть, если бы решили хранить подробную информацию о пользователях вместе с каждым из списков, иа которые они подписались. Размещение данных о пользователях в виде отдельной таблицы позволяет не тратить места впустую, а если возиикнет необходимость изменить какую-либо информацию о пользователе (скажем, при изменении адреса электронной почты), то потребуется обновить только единственную запись для всех списков рассылкн, на которые этот пользователь подписан.

Таблица Subscribers

Таблица Subscribers в основном используется для хранения связей между списком и пользователем. В ней также хранится дата, когда пользователь первый раз подписался на этот список. Данная таблица имеет следующую структуру:

Название столбца	Тип	Размер	Допускается значение null	Описание
ID	AutoNumber		Нет	Уникальный идентнфикатор подписки
ListID	Integer		Нет	Идентификатор списка, на который подписался пользователь

Продолжение таблицы

Название столбца	Тип	Размер	Допускается значение null	Описание
UserID	Integer		Нет	Идентификатор пользователя, который должен совпадать с идентификатором пользователя в таблице UserDataTable
CreateDate	Date		Нет	Дата создания данной подписки

Таблица NewsHistory

Таблица NewsHistory используется для хранения предыдущих сообщений, отправленных подписчикам всех рассылочных списков, и имеет такую структуру:

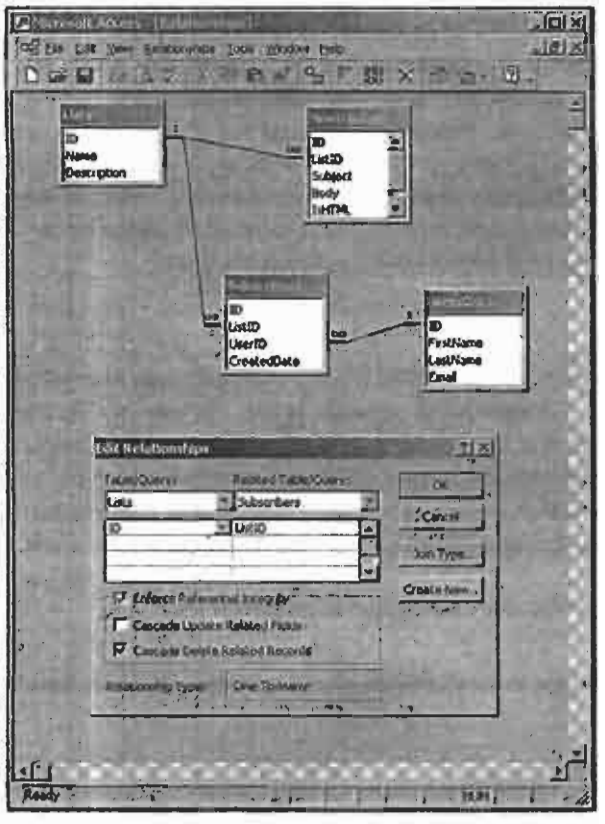
Название столбца	Тип	Размер	Допускается значение null	Описание
ID	AutoNumber		Нет	Уникальный идентификатор сообщения
ListID	Text		Нет	Название списка рассылки, в соответствии с которым рассылалось данное сообщение
Subject	Text	200	Нет	Предмет сообщения
Body	Memo		Да	Тело сообщения
IsHTML	Boolean		Нет	Если значение равно true, это означает, что сообщение отправлялось в HTML-формате
SentDate	Date		Нет	Дата отправки сообщения

Отношения между таблицами

Если вернуться немного назад и взглянуть на рисунок, на котором представлены отношения между таблицами, то можно увидеть, что между таблицами Lists и Subscribers и Lists и NewsHistory существует отношение один ко многим, так как каждый список может иметь несколько подписчиков и несколько отправленных ранее сообщений. Аналогичным образом, один пользователь может подписаться более чем на один список, поэтому отношение между таблицами UsersData и Subscribers — также один ко многим.

Задать отношения между таблицами с помощью Access очень просто. Пройдем эту процедуру последовательно, шаг за шагом. Откройте файл NewsMailer.mdb и выберите пункт меню Tools | Relations. Появится диалоговое окно, в котором на переднем плане изображены четыре используемые таблицы, а на заднем — пустое окно. Последовательно выберите и добавьте в это окно одну за другой все таблицы, изображенные в первом диалоге, после чего закройте окно. Теперь в окне, которое изображалось на заднем плане, выводятся все четыре таблицы, и можно приступать к созданию отношений между ними. Сначала нужно привязать к таблице Lists таблицу Subscribers. Щелкните мышью на поле ListID таблицы Subscribers, перенесите его в поле ID таблицы Lists и отпустите. Это приведет к появлению диалогового окна, изображенного на рисунке на стр. 801.

Два раскрывающихся списка в этом диалоговом окне позволяют выбирать поля, между которыми устанавливаются отношения. Если была проведена операция переноса мышью, то необходимые поля уже являются выбранными, а описание,



расположенное в нижней части диалога, гласит "один ко многим", что означает, что один список рассылки предназначается для множества подписчиков. Вместо операции переноса мышью можно было щелкнуть мышью два раза на основном окне (с таблицами), что привело бы к переходу в то же диалоговое окно, но только без наличия выбранных полей таблиц, между которыми устанавливаются связи.

Прежде чем окончательно создать отношение, требуется учесть еще один момент: необходимость установки Referencial Integrity (ссылочная целостность) и включения Cascade Delete (каскадное удаление), как это показано на рисунке. Поступив таким образом, мы добьемся того, что при удалении некоторого списка привязанные к нему подписчики также будут автоматически удалены. Есть еще опция Cascade Update (каскадное обновление), которая гарантирует, что все значения `ListID` в таблице `Subscribers` будут обновляться при изменении значений `ID` в таблице `Lists`, однако нам такой режим не требуется, поскольку поле `ID` имеет значение типа `AutoNumber`, которое не может изменяться после включения записи в таблицу.

Теперь нам следует создать отношения между таблицами `Lists`—`NewsHistory` и `Subscribers`—`UsersData` аналогичным образом, определяя, между какими полями должны быть установлены связи с помощью вышеприведенного рисунка.

Бизнес-классы, предназначенные для доступа в базе данных

Код, который осуществляет работу с базами данных, логически и физически подразделяется на несколько различных классов. Все эти классы будут откомпилированы в библиотеку классов с именем `NewsMailerDB`. Остальная часть кода — классы кода поддержки приложения — будет помещена в отдельный модуль. Это наилучший подход, ибо, если возникает необходимость внести какие-либо изменения в работу базы данных или требуется установить новую версию SQL Server, для этого будет нужно внести изменения и повторно откомпилировать только бизнес-модуль. Это лучше, чем компилировать все вместе (бизнес-классы и класс поддержки) в единый модуль, поскольку можно использовать бизнес-модуль самостоятельно, например, в клиентской части приложения Windows.

Для данного приложения такой подход не дает существенных преимуществ; структура приложения является довольно простой, компилировать приходится не такие уж большие файлы, и никакие другие клиенты эти бизнес-классы не используют. Поэтому, даже если мы все откомпилируем в единую DLL, это не приведет к серьезным проблемам. Но мы будем использовать отдельные DLL-модули, для того чтобы продемонстрировать, насколько все это просто, и для того, чтобы приучить вас к хорошим манерам!

Если вы разрабатываете проект с нуля, вы должны посвятить все время, затраченное на создание проекта, объектно-ориентированной разработке, для чего обычно следует отделить бизнес-классы от кода пользовательского интерфейса и компилировать их в разные библиотеки. Разбиение кода на несколько отдельных модулей может быть осуществлено и позднее, поскольку это относится к процедуре компиляции. Однако при этом необходимо создавать такие классы, единственная задача которых — осуществлять взаимодействие с базой данных, размещая весь код, ответственный за пользовательский интерфейс, в других классах.

Создание проекта

Если для разработки используется Visual Studio.NET, то следует создать новый проект с типом C# | Class Library и назвать его `NewsMailerDB`. Затем необходимо приступить к созданию файлов классов для данного модуля, что будет объясняться в следующих разделах. Необходимо создать новый файл для каждого класса с помощью команды Project | Add New Class, либо с помощью Project | Add Existing Item, если вы пожелаете импортировать готовые файлы с web-сайта издательства Wrox.

Обратите внимание на то, что код этих бизнес-классов воспроизводится в тексте книги.

Класс ListsDB

Класс `ListsDB` предоставляет все функциональные возможности, необходимые как для выполнения манипуляций над уже существующими списками рассылки, так и для создания новых. Информация об этом классе сведена в следующую таблицу:

Класс ListsDB Метод	Описание
`public ListsDB(string ConnString)`	Конструктор класса, которому в качестве параметра передается строка соединения, описываемая в установках файла (к ним обсуждению мы перейдем позже)
`public DataSet GetLists()`	Возвращает объект `DataSet`, в котором содержатся все доступные списки новостей
`public DataRow GetDetails(int ListID)`	Позволяет получать подробную информацию о списке по заданному идентификатору
`public bool Add(string ListName, string listDescr)`	Добавляет новый список с заданным именем и описанием
`public bool Delete(int ListID)`	Удаляет указанный список
`bool Update(int ListID, string ListName, string listDescr)`	Позволяет изменять свойства `Name` и `Description` указанного списка

Метод `GetLists()` возвращает набор данных, в котором находится все содержимое таблицы `Lists`, то есть все доступные списки и их свойства. Этот метод возвращает единственный список, который задается идентификатором, передаваемым методу в качестве параметра.

С помощью метода `Delete()` можно удалить запись с идентификатором, передаваемым этому методу в качестве параметра. Почти так же работают методы `Add()` и `Update()` за исключением того, что параметры могут содержать еще и данные, которые должны быть модифицированы или добавлены.

Класс SubscribersDB

Класс `SubscribersDB` предназначается для доступа к таблицам `Subscribers` и `UsersData`. Он обладает структурой, аналогичной структуре класса `ListsDB`, и хранится в файле `Subscribers.cs`. Все методы этого класса сведены в следующую таблицу:

Класс Subscribers DBМетод	Описанне
`public SubscribersDB(string ConnString)`	Конструктор класса, которому в качестве параметра передается строка соединения, описываемая в установках файла
`public DataSet GetSubscribers(int ListID)`	Возвращает список подписчиков данного списка
`public DataRow GetDetails(int SubscrID)`	Позволяет получать подробную информацию о списке по заданному идентификатору
`public bool Add(int ListID, string FirstName, string LastName, string Email)`	Добавляет нового подписчика
`public bool Delete(int SubscrID)`	Удаляет подписчика
`public bool Update(int SubscrID, string FirstName, string LastName, string Email)`	Позволяет обновить подробную персональную информацию о подписчике
`public int GetUserID(int SubscrID)`	Возвращает идентификатор пользователя по заданному идентификатору подписки
`public int GetUserID(string Email)`	Возвращает идентификатор пользователя по заданному адресу электронной почты
`public int GetSubscriberID(int ListID, string Email)`	Возвращает идентификатор подписчика для конкретного списка и конкретного пользователя, определяемый по уникальному адресу электронной почты

Этот класс оказывается несколько более сложным, чем класс `ListsDB`, поскольку ему приходится иметь дело с двумя таблицами и управлять отношениями между ними. Функция `GetSubscribers()` возвращает всех подписчиков того списка, идентификатор которого передан ему в качестве параметра. В нем используется SQL-оператор, в котором содержится команда `INNER JOIN`, позволяющая получать информацию сразу из двух таблиц — `UsersData` и `Subscribers`:

```
public DataSet GetSubsctibers (int ListID)
{
    string sqlCmd = "SELECT FirstName, LastName, Email, ";
    sqlCmd += "ListID, UserID, CreatedDate, Subsctibers.ID FROM UsersData ";
    sqlCmd += "INNER JOIN Subscribers ON UsersData.ID= Subsctibers.UserID ";
    sqlCmd += "WHERE Subscribers.ListID = " + ListID;

    DataSet subsctibers = new DataSet();
    OleDbConnection myConn = new OleDbConnection(connString);
    OleDbDataAdapter myDAdp = new OleDbDataAdapter(sqlCmd, myConn);

    myDAdp.Fill(subscribers, "Subscribers");
    return subscribers;
}
```

Метод `GetDetails()` практически полностью аналогичен соответствующему методу класса `ListsDB`; отличие заключается в SQL-строке, которая у него такая же, как приведенная выше.

Методы Add(), Delete() и Update() позволяют вставлять, удалять и модифицировать подписку пользователей в новые списки. Они выполняют эту работу несколько по-другому, чем соответствующие методы класса ListsDB. Функция Delete() сначала удаляет подписчика из указанного списка, а затем производит проверку, включен ли данный пользователь в какие-либо другие списки. Если нет, то информация о пользователе удаляется из таблицы UsersData. Функция Add() не только добавляет подписчика в указанный список, но и проверяет, не совпадает ли переданный ей адрес электронной почты с каким-либо адресом, уже имеющимся в таблице UsersData. Если нет, то делается предположение, что это новый пользователь, и он добавляется в таблицу. Для создаваемого приложения мы будем исходить из предположения, что каждое вхождение поля электронной почты представляет уникального пользователя.

Далее приведены две перезагрузки метода GetUserID(), которые предназначаются для внутреннего использования методами Delete() и Add(). Первый вариант принимает в качестве параметра идентификатор подписчика и возвращает идентификатор пользователя, привязанный к данной подписке; этот метод используется внутри метода Delete(). Вторая перезагрузка принимает в качестве параметра строку адреса электронной почты и возвращает идентификатор пользователя, привязанный к данному адресу, если таковой существует; этот метод используется внутри метода Add().

Последняя функция данного класса — GetSubscriberID() — возвращает идентификатор подписчика определенного списка по указанному адресу электронной почты. Эта функция используется методом Add(), чтобы определить, существует ли уже пользователь, который осуществляет подписку на данный список, в таблице UsersData, или нет.

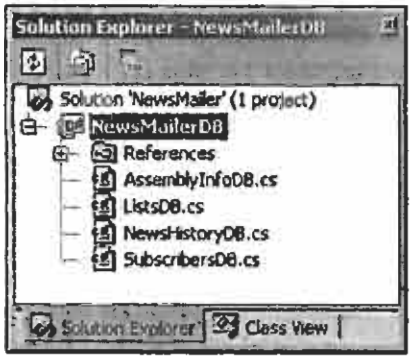
Класс NewsHistoryDB

Класс NewsHistoryDB (который хранится в файле NewsMailerDB.cs) является самым простым из всех; он состоит всего лишь из четырех методов и параметризованного конструктора метода, как показано в приведенной ниже таблице:

Класс NewsHistoryDB Метод	Описание
`public NewsHistoryDB(string ConnString)`	Конструктор класса, которому в качестве параметра передается строка соединения, которая описывается в установках файла
`public DataSet GetNews(int ListID)`	Возвращает все наименования новостей для указанного списка
`public DataRow GetDetails(int NewsID)`	Позволяет получать строку, которая идентифицирует указанную запись News
`public bool Add(int ListID, string Subject, string Body, bool IsHTML)`	Добавляет подробную информацию о записи с новостями в заданный список
`public bool Delete(int NewsID)`	Удаляет указанную запись News

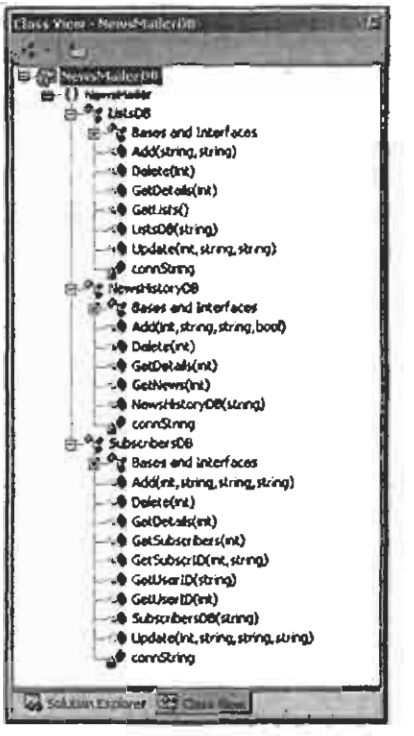
Компиляция модуля

Когда создание всех файлов завершено, можно откомпилировать модуль. Если вы выполняли все действия в Visual Studio.NET, то ваш Solution Explorer должен принять вид, изображенный на рисунке на стр. 805.



Файл `AssemblyInfo.cs` содержит множество всех атрибутов, которые транслируются в модуль наряду с исходными файлами. Этот файл создается Visual Studio.NET по умолчанию в момент создания проекта. Мы не будем вносить в него какие бы то ни было изменения.

При этом в режиме Class View должна выводиться картинка, представленная на рисунке справа.



Компиляция модуля заключается в использовании команды **Build** в VS.NET.

Управление конфигурацией данных

Программа рассылки новостей может быть настроена со специальными подтверждениями, именем и электронным адресом администратора, и некоторыми другими опциями (со всеми возможными опциями мы вскоре познакомимся). Использование жестко заданных в программе значений этих опций является плохим подходом, так как при возникновении у администратора необходимости изменить некоторую опцию (например, свой адрес электронной почты), ему придется вносить соответствующие изменения в исходный код, компилировать весь проект заново и загружать новый модуль. Это серьезная проблема, особенно если администратор сайта не является программистом. Поэтому мы будем хранить все установки в виде внешнего файла, из которого их будет очень просто считывать и записывать, а также выводить на страницу, позволяя администратору с легкостью изменять любые значения без повторной компиляции и перезагрузки.

XML-файл установок

Все установки обычно хранятся в базе данных или в текстовом файле. Если выбрать базу данных, то потребуется завести целую таблицу для хранения всего лишь нескольких установок. Можно бы хранить все установки либо в одной строке (со многими столбцами), либо по одной установке в каждой строке. Такое решение оказывается весьма простым — просто еще одна таблица и еще один класс в модуле `NewsMailerDB`, однако оно означает неэффективное использование ресурсов базы данных. Хорошим решением является использование XML-файла, который обладает вполне подходящим форматом для хранения установок приложения. В .NET Framework также предусмотрены классы, позволяющие легко считывать и записывать все установки и представлять их в виде свойств класса несколькими строками кода. Этот код будет рассмотрен ниже, а пока начнем с объяснения самих установок.

Файл `NewsMailer.Settings` используется для хранения строки соединения с базой данных всех установок, а также приложения за исключением касающихся внешнего представления приложения. Все возможные установки перечислены в приводимой ниже таблице.

Установки `NewsSubject`, `[Un]SubscriptionSubject` и `[Un]SubscriptionMessage` могут содержать в себе специальные теги, например, `#SubscriberFirstName#` или

#ListDescription#, которые будут заменяться на соответствующие им значения перед отправкой каждого сообщения. Позднее в настоящем разделе мы детально разберем эту возможность.

Название установки	Тип	Допускается значение null	Описание
SenderName	string	Да	Имя отправителя новостей
SenderEmail	string	Нет	Адрес электронной почты отправителя новостей
NewsSubject	string	Нет	Предмет новостного сообщения, использующийся по умолчанию. У администратора есть возможность создать сообщение, добавив произвольный текст к этой строке непосредственно из страницы
Signature	string	Да	Подпись, которая будет автоматически добавляться в конец сообщения
SubscriptionSubject	string	Нет	Предмет сообщения, отправляемого по электронной почте в качестве подтверждения успешной подписки
SubscriptionMessage	string	Да	Тело сообщения, отправляемого по электронной почте в качестве подтверждения успешной подписки
UnSubscriptionSubject	string	Нет	Предмет сообщения, отправляемого по электронной почте в качестве подтверждения успешного отказа от подписки
UnSubscriptionMessage	string	Да	Тело сообщения, отправляемого по электронной почте в качестве подтверждения успешного отказа от подписки
SendSubscriptionEmail	bool	Нет	Если это значение равно true, то новые подписчики будут получать по электронной почте подтверждение о том, что подписка прошла успешно
SendUnSubscriptionEmail	bool	Нет	Если это значение равно true, то существующие подписчики будут получать по электронной почте сообщение о том, что их подписка завершена
SubscribeURL	string	Нет	Полный URL страницы Subscribe.ASP.NET, который используется для добавления или удаления пользователя из списка

В этой таблице приводятся различные типы установок; все установки хранятся в XML-файле в виде строк. Этот тип обозначает тип свойства класса, которому (свойству) соответствует значение данной установки. Столбец "Допускается значение null" имеет аналогичный смысл: в нем описываются установки, которые могут отсутствовать в файле установок. Не существует никакого файла со схемой, которая позволяла бы получать подтвержденне того, что та или иная установка на самом деле присутствует в файле установок — страница ASP.NET, которая используется для их считывания и изменения, отвечает за сохранение допустимости структуры файла.

XML-файл выглядит приблизительно следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<AppSettings>
   <SenderName>NewsMailer Team</SenderName>
   <SenderEmail>admin@newsmailerserver.com</SenderEmail>
   <NewsSubject>News 4 you: #ListName#</ NewsSubject>
   <Signature>The NewsMailer Team</Signature>
   <SubscrSubject>Welcome to the mailing list #ListName#</SubscrSubject>
   <SubscrMessage>Welcome dear subscriber</SubscrMessage>
   <UnSubscrSubject>Bye bye from #ListName#</UnSubscrSubject>
   <UnSubscrMessage>You've been removed from #ListName#</UnSubscrMessage>
   <SendSubscrEmail>true</SendSubscrEmail>
   <SendUnSubscrEmail>true</SendUnSubscrEmail>
   <SubscribeURL>http://localhost/newsmailer/subscribe.aspx</SubscribeURL>
</AppSettings>
```

Модуль NewsMailerConfig

Сериализация объекта — это процесс сохранения состояния данного объекта в памяти или в виде постоянного двоичного или ASCII-файла. Под состоянием объекта понимается множество всех его свойств, значений и ссылок на другие объекты. Напротив, десериализация приводит к созданию нового объекта, состояние которого определяется данными, хранящимися в некотором файле. .NET Framework существенно упрощает выполнение сериализации и десериализации за счет множества классов, которые позволяют работать с несколькими различными форматами хранения.

Если вы используете VS.NET, создайте для начала новый проект типа `Class Library` и переименуйте установленный по умолчанию файл класса в `AppConfig.cs`. Если вы набираете текст с помощью текстового редактора, то сохраните этот файл в директории `NewsMailer\Config`. Ниже приводится сводная таблица класса `AppSettings`, который отвечает за создание XML-файлов, подобных приведенному выше, из сериализованных экземпляров. Этот файл состоит из множества общих свойств, имена которых совпадают с именами соответствующих свойств из XML-файла.

Все свойства класса `AppSettings` помечены атрибутом `XmlElement`. Это сделано именно так, поскольку требуется, чтобы все эти свойства могли быть сериализованы в виде XML, а для этого используется класс `XmlSerializer` из пространства имен `System.Xml.Serialization`. Поскольку класс `AppSettings` не обладает методами, позволяющими ему сериализовать и десериализовать себя, эта задача возлагается на класс поддержки `AppConfig`, также хранящийся в файле `AppConfig.cs`. Вот содержимое этого класса:

Класс AppConfig Метод	Описание
`public static AppSettings GetSettings(string fileName)`	Находит XML-файл, в котором хранятся все установки, и создает объект класса `AppSettings`, содержащий эти данные
`public static void SaveSettings(string fileName, AppSettings data)`	Получает объект `AppSettings` и записывает хранящиеся в нем данные в указанный XML-файл

Эти два метода достаточно сложны, поэтому мы объясним их более подробно. Для начала приведем код метода `GetSettings()`:

```
public static AppSettings GetSettings(string fileName)
{
   HttpContext = HttpContext.Current;

   AppSettings data = (AppSettings)context.Cache["AppSettings"];

   if (data == null)
   {
      XmlSerializer serializer = new XmlSerializer(typeof(AppSettings));
      try
      {
         // создать файловый поток для чтения XML-документа
         FileStream fs = new FileStream(filename. FileMode,Open);
         XmlReader = new XmlTextReader(fs);

         // посмотреть состояние объекта
         data = (AppSettings)seriallizer.Deserialize(reader);
         reader.Close();

         context.Cache.Insert("AppSettings", data,
         new CacheDependency(filename));
      }
      catch (System.IO.FileNotFoundException)
      {
         // если файл не найден, вернуть новый пустой класс
         data = new AppSettings();
      }
   }
   return data;
}
```

Прежде всего данный метод осуществляет проверку, не находится ли объект с данными установками в кэш-памяти, куда он попадает при первом обращении к методу. Если объект уже находится в кэш-памяти, то метод `GetSettings()` просто возвращает эти данные. Если объект в кэш-памяти отсутствует, то сначала создается объект `XmlSerializer`, затем осуществляется открытие параметра файла, указанного в качестве первого, после чего для потока файла создается объект `XmlTextReader`. В завершение состояние объекта извлекается посредством метода `Deserialize()` объекта `XmlSerializer`, а полученный в результате всех этих действий объект `AppSettings` помещается в кэш-память для последующего использования.

В этой программе есть несколько примечательных моментов. Первый заключается в том, что путь к файлу с установками не "зашит" в программу, а передается в качестве параметра. Это необходимо, поскольку данный код будет откомпилирован в модуль и может быть использован произвольным клиентом, поэтому файл с установками может располагаться в различных местах.

Вторым любопытным моментом является способ, с помощью которого установки переносятся в кэш. Обратите внимание на следующую строку кода:

```
Context.Cache.Insert("AppSettings", data, new CacheDependency(fileName));
```

Третий параметр означает, что информация, хранящаяся в кэш-памяти, зависит от файла, который передается в качестве параметра. Если этот файл обновляется, то информация, хранящаяся в кэш-памяти, более не считается правильной, поэтому при повторном обращении к установкам необходимо выполнить десериализацию файла еще раз. Тот факт, что состояние файла отслеживается автоматически,

является замечательной особенностью кэширования в ASP.NET, которая позволяет вносить изменения в содержимое файла установок с помощью произвольного редактора, перезагружать файл и при этом знать, что все установки будут перезагружены в кэш-память.

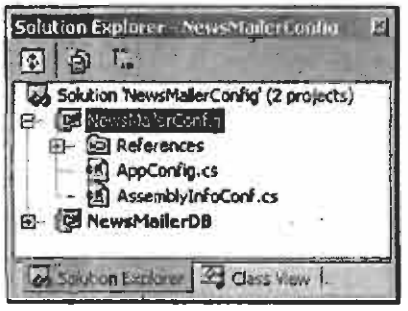
И последний момент: когда состояние объекта перезагружается из файла, то все установки, хранящиеся в XML-файле в виде строк, автоматически приводятся к тому типу, которым обладает соответствующее свойство.

Метод `SaveSettings()` обладает структурой, аналогичной структуре метода `GetSettings()`:

```
public static void SaveSettings(string FileName, AppSettings data)
{
   XmlSerializer = new XmlSerializer(typeof(AppSettings));

   // Создать XmlTextWriter используя FileStream
   FileStream fs = new FileStream(fileName, FileMode.Create);
   XmlTextWriter writer = new XmlTextWriter(fs, new UTF8Encoding());

   // Осуществить сериализацию, используя XmlTextWriter
   serializer.Serialize(writer, data);
   writer.Close();
}
```

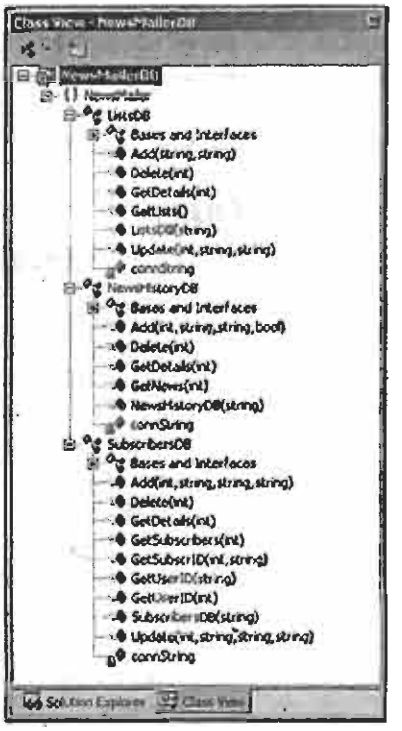


Единственный новый момент заключается в том, что некоторые символы, например '<' и '>', кодируются, поскольку с точки зрения XML они обладают особым смыслом (они используются для обрамления XML-тегов). Способ такого кодирования задается в конструкторе объекта `XmlTextWriter`.

Компиляция модуля

Если вы вели разработку в Visual Studio.NET, то ваш Solution Explorer должен принять вид, изображенный на рисунке слева.

При этом в режиме Class View должна выводиться следующая картинка, представленная на рисунке справа.

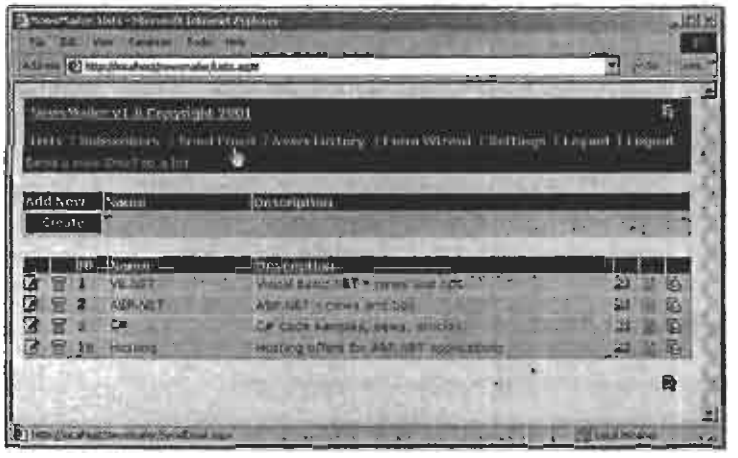


Управление пользовательским интерфейсом

Разработка базы данных и бизнес-классов завершена, т. е. бóльшая часть работы уже выполнена. Теперь мы собираемся заняться разработкой пользовательского интерфейса приложения, а также использовать бизнес-классы для чтения и записи данных из базы данных.

Прежде чем переходить к рассмотрению кода на ASP.NET и C#, взглянем на некоторую часть результатов, к которым мы стремимся (см. рис. на стр. 810).

Страница, изображенная на указанном рисунке, позволяет администратору просматривать имеющиеся в наличии списки рассылки, создавать, уничтожать или редактировать эти списки, а также переходить к списку подписчиков или к другим страницам приложения. В таблице на данной странице показаны все списки, которые были получены из базы данных, в ней также выводятся



их свойства и предусматривается возможность их сортировки по значению одного из следующих свойств: `ID`, `Name`, `Description`. Все иконки, выводящиеся в таблице (рисунки, расположенные левее столбца `ID` и правее столбца `Description`), представляют собой гиперссылки на другие страницы, позволяющие выполнять какие-либо другие действия над данным списком или над данной строкой. Ниже приводится перечень иконок, которые могут располагаться внутри таблицы с описанием выполняемых ими действий.

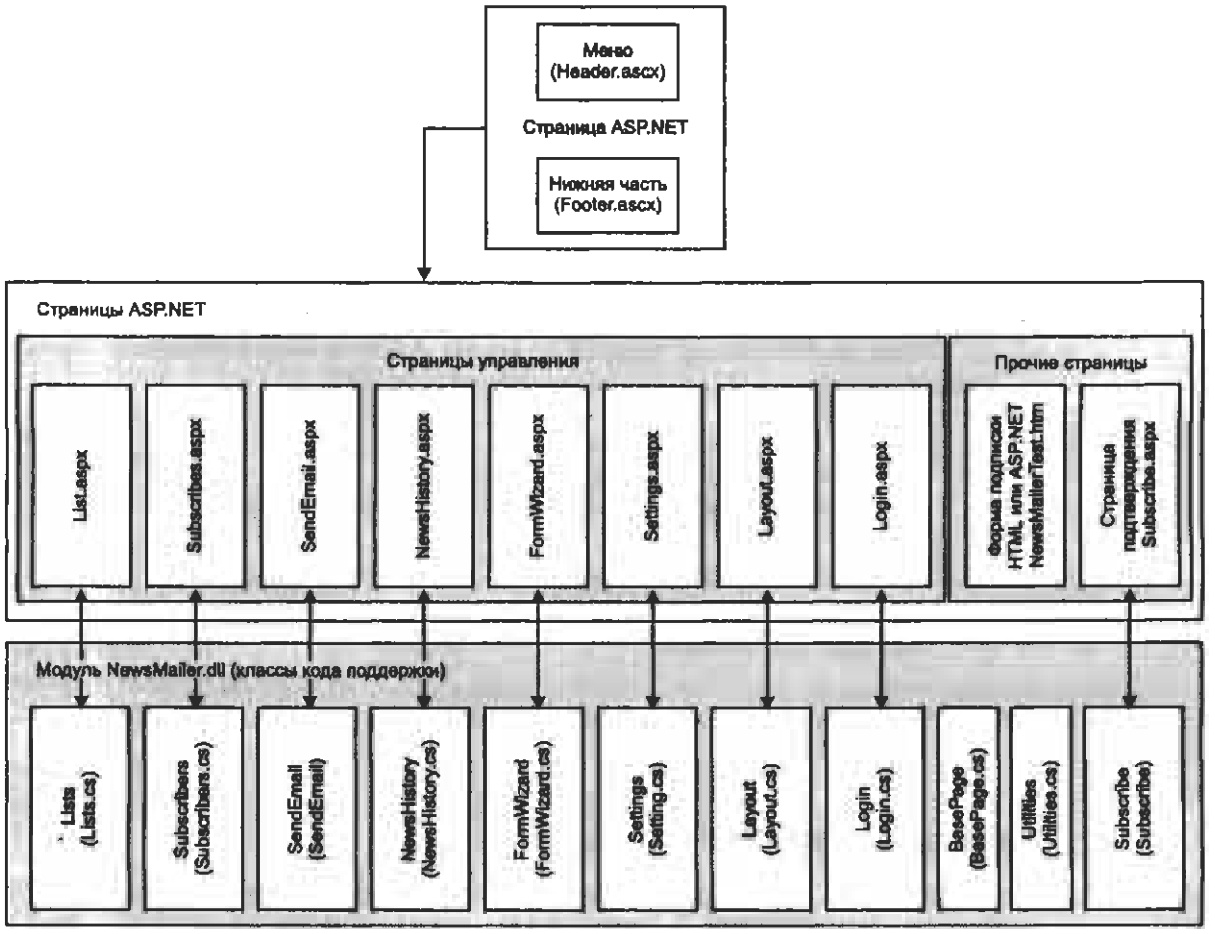
Иконка	Описание
	Редактирование записи
	Удаление записи
	Переход на страницу, на которой выводятся подписчики данного списка рассылки
	Переход на страницу, на которой выводятся прошлые сообщения, отправленные в соответствии с данным списком рассылки
	Переход на страницу, на которой составляется и отправляется новое сообщение подписчикам, входящим в данный список рассылки

Если вы когда-либо программировали на обычном ASP, то вы, вероятно, думаете, что для этой таблицы потребуется большой объем сложного кода, позволяющего управлять гиперссылками, рисунками, изменением цвета строк, сортировкой данных и т. п. Но вы удивитесь, насколько просто всего этого можно добиться с помощью ASP.NET и нескольких строк кода на C#.

Все приложение состоит из нескольких страниц ASP.NET, большинство из которых относится к административному разделу. На диаграмме (см. стр. 811) приводится стандартное расположение страницы, а также полный список всех страниц, имен файлов, в которых они хранятся, и соответствующих файлов и классов с кодом поддержки.

Создание пользовательских управляющих элементов

Если есть некоторые составные части пользовательского интерфейса, которые требуется использовать более, чем на одной странице, то следует подумать о том, каким образом можно использовать код повторно без копирования и переноса его из одной страницы в другую. В данной программе мы хотим иметь возможность общего использования заголовка и нижней части страницы, которые соответственно представляют собой верхнее меню и маленькую иконку, расположенную внизу справа непосредственно под таблицей и позволяющую возвращаться на начало страницы. В традиционном ASP потребовалось бы включить два отдельных файла посредством директивы `<!--#INCLUDE -->`, однако в ASP.NET есть лучшая альтернатива — пользовательские управляющие элементы. С их помощью можно выделить общий код в отдельный файл, который может быть в дальнейшем использован везде, где в этом возникнет необходимость, путем присваивания имени импортируемому управляющему элементу, изменения его общих свойств или вызова общих



методов, которыми он обладает. Другими словами, это не просто участок кода, вставляемый в нашу страницу, он представляет собой графический компонент, обладающий собственным именем и атрибутами. Являясь серверным компонентом, он, как и другие серверные компоненты ASP.NET, обрабатывается на сервере, а все, что получает браузер пользователя, — это стандартный HTML, совместимый с любым браузером.

В нашем проекте используются два пользовательских управляющих элемента:

Пользовательский управляющий элемент	Описание
`Header.ascx`	Осуществляет добавление заголовка, общего для всех страниц web-сайта. Этот управляющий элемент состоит из трех частей: • Раздел заголовка, представленного в виде таблицы, в которой хранятся текст и логотип. • Еще одна таблица, в которой хранятся ссылки на другие web-страницы. • Надпись, представляющая собой текст, который описывает каждую из ссылок, когда на нее указывает курсор мыши.
`Footer.ascx`	Содержит всего лишь образ ссылки, которая переносит пользователя обратно на начало страницы.

Использование пользовательских управляющих элементов на web-странице

Чтобы использовать управляющий элемент из web-страницы, необходимо зарегистрировать его из этой страницы, указав имя исходного файла и префикс, позволяющий отличать его от других серверных управляющих элементов. Следующие две строки должны быть включены в начало страницы:

```
<%@ Register TagPrefix="NewsMailer" TagName="Header" src="Header.ascx" %>
<%@ Register TagPrefix="NewsMailer" TagName="Footer" src="Footer.ascx" %>
```

Ниже приводится код, который позволяет добавить пользовательские управляющие элементы `Header` и `Footer` на обычную страницу:

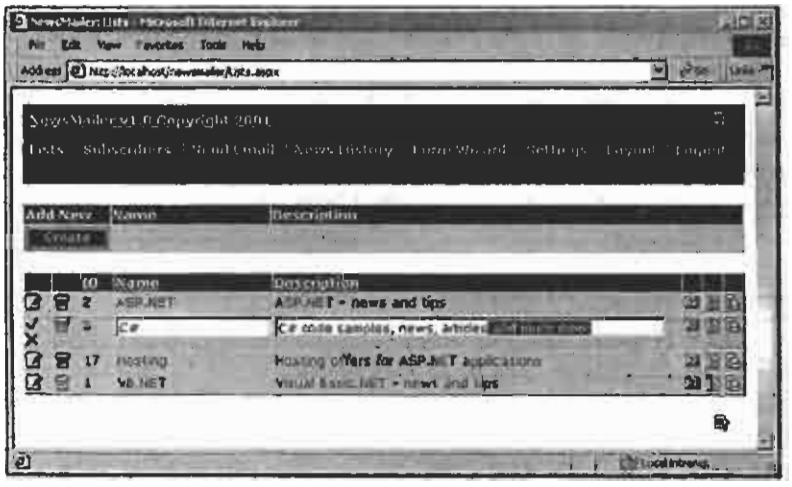
```
<html>
<head>
   <title>NewsMailer: Subscribers</title>
   <link rel="stylesheet" HREF="Styles.css" TYPE="text/css">
</head>
<body>
   <!-- Insert the menu user control -->
   <NewsMailer:Header id="Menu" runat="server" />

   <b> Some HTML code</b>

   <!-- Insert the footer -->
   <NewsMailer:Footer id="Footer" runat="server" />
</body>
</html>
```

Страница управления списками

На ранее представленном рисунке страница управления списками рассылки изображена в том виде, в каком она появляется на экране сразу после загрузки. Однако ее внешний вид может претерпевать определенные изменения в зависимости от предпринимаемых действий. Вот что вы увидите, если нажмете на иконку с изображением блокнота и ручки, расположенную у самого левого края, для того чтобы отредактировать список (см. рис. слева).



Свойства `Name` и `Description` текущей страницы выводятся внутри двух текстовых окон, и у администратора есть возможность изменить эти значения, после чего щелкнуть мышью либо на зеленой иконке с галочкой, чтобы сохранить эти изменения в базе данных, либо на X, для того чтобы отказаться от этой операции. Обратите внимание на то, что столбец `ID` не подлежит редактированию, поскольку он используется в качестве счетчика и его значение не может быть изменено, а также на то, что списки отсортированы по имени, а не по `ID`, как прежде. Для сортировки необходимо щелкнуть мышью на заголовке того столбца, по значениям которого вы хотите произвести сортировку.

Для того чтобы добавить новый список, администратор должен щелкнуть мышью на кнопке **Button**, расположенной слева вверху основной страницы. Вот к какому результату это приведет:



На экран выводятся два пустых текстовых окна, и администратор заполняет их, вводя названия и описание того списка рассылки, который он собирается создавать. Выполнение этой операции может быть подтверждено или отменено посредством двух иконок, расположенных с левой стороны. В обоих случаях (это также справедливо и для редактирования) текстовые окна убираются, а картинка возвращается к своему первоначальному состоянию. Если нажать иконку для редактирования новой строки, то такой эффект достигается почти автоматически, и тогда при добавлении новой строки приходится обрабатывать события управляющих элементов для вывода или свертывания текстовых окон самостоятельно. Этот процесс оказывается намного проще, чем может показаться на первый взгляд — для его осуществления необходимо написать всего лишь несколько строк кода, к чему мы перейдем буквально через минуту.

Страница Lists.aspx

Это страница ASP.NET, в которой описывается пользовательский интерфейс; в ней может также храниться код описания сценария, как и на страницах ASP. Одно из наиболее значимых преимуществ ASP.NET заключается в том, что код оказывается отделенным от пользовательского интерфейса за счет использования **файлов с кодом поддержки** — этот термин мы будем очень часто использовать по ходу настоящего раздела (и который мы уже несколько раз использовали, не давая ему определения). В файле с кодом поддержки находится один или более классов, которые обрабатывают все события, возникающие в web-странице. Разделение этих двух уровней позволяет web-дизайнеру вносить изменения во внешнее представление web-страниц без необходимости разбираться в запутанном коде на VBScript или Jscript, который распределен по всей странице вперемешку с HTML-кодом. Файлы с кодом поддержки могут быть откомпилированы в отдельный модуль, что существенно упрощает установку приложения (поскольку в этом случае приходится загружать только один DLL-файл вместо нескольких исходных файлов с кодами на C#) и повышает уровень обеспечения безопасности вашего приложения в том смысле, что если кто-то получит доступ к серверу, то он не сможет также быстро прочитать бизнес-код, как в случае с ASP.

Чтобы создать страницу `Lists.aspx` в VS.NET, выберите пункт меню **Project | Add Web Form**. При работе в любом другом редакторе просто создайте новый файл в папке `NewsMailer`.

Исходный код для этой страницы слишком велик, чтобы приводить его целиком, поэтому будем останавливаться только на тех его участках, которые представляют для нас интерес как для программистов на C#. В первой строке содержится описание того, какой базовый класс используется при обработке событий, возникающих на данной странице:

```
<%@ Page language="c#" Inherits="NewsMailer.Lists" %>
```

В этом классе содержится весь код, который выполняется при загрузке данной страницы при нажатии клавиши, а также когда мы хотим добавить, удалить или отредактировать какой-либо список рассылки. Этот класс, называющийся `Lists`, входит в состав пространства имен `NewsMailer` и компилируется в отдельный модуль `NewsMailer.dll`, к рассмотрению которого мы перейдем позднее. Страница автоматически просматривает папку с именем `bin` в корневой web-папке, находит модуль и реализует наследование от этого класса. Файл, в котором содержится исходный код на C# (в данном случае это файл `Lists.cs`), представляет собой файл с кодом поддержки.

Двигаясь далее по телу страницы, мы обнаруживаем форму с кодом для середины web-страницы, расположенной между заголовком и нижней частью страницы. Первое, на что мы обращаем внимание,— это кнопка **Create** в верхнем левом углу непосредственно под заголовком. При нажатии этой кнопки появляются два текстовых окна и две иконки — для подтверждения и для отказа от данной операции, которые были продемонстрированы ранее. Остальная часть страницы остается без изменений. Ниже приводится код для этой кнопки:

```
<asp:TableRow ID="CreateNewRow">
   <asp:TableCell ColumnSpan+ "3">
      <asp:Button runat="server" Text="Create"
                  ID="Create_Click" Width="80px"
                  OnClick="Create_Click" Width="80px"
      />
   </asp:TableCell>
</asp:TableRow>
```

Далее идет строка таблицы, по умолчанию являющаяся невидимой, но которая становится видимой при щелчке мышью на кнопке **Create**. Эта часть программы позволяет модифицировать список и принять сделанные изменения или отказаться от них:

```
<asp:TableRow ID="AddNewControlsRow" Visible="false">
   <asp:TableCell>
      <asp:Image  ImageURL="./Images/imgSpacer.gif"
                  Width="15px" Height="1px" runat="server" />
      <asp:LinkButton ID="AddNew" runat="server" />
                  Text="<img beorder=0 scr=./Images/imgOK.gif
                  Alt='Add new list'>"
                  OnClick-"AddNew_Click"
      />
      <asp:Image  ImageURL="./Images/imgSpacer.gif"
                  Width="15px" Height="1px" runat="server" />
      <asp:LinkButton ID="CancelAddNew"
                  runat="server" CausesValidation="false"
                  Text="<img beorder=0 scr=./Images/imgCancel.gif
                  Alt='Cancel editing mode'>"
                  OnClick-"CancelAddNew_Click"
      />

   </asp:TableCell>
   </asp:Table Cell VerticalAlign="Top">
      <asp:TextBox runat="server" Class="TextBox"
                   ID="NewListName" Width="150px" />
      <asp:RequiredFieldValidator Id="ValidateNewName"
                   runat="server" COntrolToValidate="NewListName"
                   Display="dynamic"<br>" Name is required
      </asp:RequiredFieldValidator>
   </asp:TableCell>
```

```
        </asp:Table Cell VerticalAlign="Top">
            <asp:TextBox runat="server" Class="TextBox"
                         ID="NewListDescr" Width="450px" />
        </asp:TableCell>
    </asp:TableRow>
```

Обратите внимание на то, что все серверные управляющие элементы, к которым требуется обеспечить доступ программным путем, обладают атрибутом `ID`, а весь код располагается внутри серверной формы. Также обратите внимание на то, что при создании нового списка рассылки задание его имени является необходимым, что обеспечивается управляющим элементом `RequiredFieldValidator`. Назначение свойства `ControlToValidate` (управляющий элемент, подлежащий проверке) очевидно, а свойство `Display`, которому присваивается значение `dynamic`, означает, что область для сообщения об ошибке будет выделяться в динамическом режиме.

Одним из наиболее интересных серверных управляющих элементов является элемент табличного вывода, который в нашем приложении используется для вывода списков рассылки. Этот управляющий элемент обладает широким спектром функциональных возможностей, для использования которых требуется всего лишь несколько строк кода: отличающиеся стили для подлежащих изменению строк, сортировка данных и постраничное разбиение данных. Однако наиболее важным представляется то, что этот управляющий элемент является полностью настраиваемым и основанным на использовании шаблонов. С помощью программного шаблона можно определить содержимое каждого столбца и указать способ, которым эти данные должны выводиться.

Именно здесь задаются свойства управляющего элемента для табличного вывода данных:

```
<asp:DataGrid id="ListsGrid" runat="srver"
      CssClass="Grid_General"
      HeaderStyle-CssClass="Grid_Header"
      ItemStyle-CssClass="Grid_Item"
      AlternatingItemStyle-CssClass="Grid_AlternatingItem"
      AllowSorting="True"
      AutoGeneratedColumns="False"
      DataKeyField="ID"
      OnEditCommand="ListsGrid_Edit"
      OnCancelCommand="ListsGrid_CancelEdit"
      OnUpdateCommand="ListsGrid_Update"
      OnDeleteCommand="ListsGrid_Delete"
      OnSortCommand="ListsGrid_Sort" >
```

Задаются стили, которые будут использоваться при выводе различных составляющих таблицы, кроме того, мы определяем, что эти данные могут быть отсортированы, а также задаем столбец в исходной таблице, который будет использоваться в качестве первичного ключа данных таблицы (свойство `DataKeyField`). В последних строках описываются процедуры обработки событий, вызываемые для реализации различных операций над таблицей.

Далее требуется объявить, каким образом будет представлен каждый столбец. Нам необходимы следующие столбцы:

- Допускающий щелчок мышью рисунок для редактирования строки
- Допускающий нажатие мышью рисунок для удаления строки
- Столбец для вывода названия списка
- Столбец для вывода описания списка

- Допускающий щелчок мышью рисунок для перехода к странице, где выводятся подписчики списка, который находится в текущей строке
- Допускающий щелчок мышью рисунок для перехода к странице, на которой выводятся ранее отправленные сообщения для данного списка
- Допускающий щелчок мышью рисунок для перехода к странице, которая позволяет отправить новое сообщение по электронной почте всем подписчикам данного списка

Поскольку табличное представление данных до сих пор в книге не встречалось, то мы подробно остановимся на его реализации. Ниже приводится код, необходимый для вывода первых двух столбцов:

```
<Columns>
   <asp:EditCommandColumn
         ItemStyle-Width="25px"
         EditText="<img border=0 Alt='Edit this list'
                   src=./Images/imgEdit.gif>"
         CancelText="<img border=0 src=./Images/imgCancel.gif>"
         UpdateText="<img border=0 src=./Images/imgOK.gif>"
   />
   <asp:ButtonColumn
         ItemStyle-Width="25px"
         Text="<img border=0 Alt='Delete this list'
               src=./Images/imgDelete.gif>"
         CommandName="delete"
   />
```

`EditColumnButton` — это особый столбец, содержащий ссылку на начало редактирования, а `ButtonColumn` позволяет создавать обычные кнопки или ссылки. Для выполнения любой операции, которая может потребоваться, необходимо обрабатывать событие `Click`. Обратите внимание на то, что для представления кнопок свойство столбца `Text` задается с помощью тега `<img>`. Если включить в программу обычную строку вместо этой HTML-строки, то мы получаем стандартную кнопку с надписью в виде этой строки.

Ниже приводится программа для вывода столбцов, содержащих поля списка `ID`, `Name` и `Description`:

```
<asp:BoundColumn HeaderText="ID" ItemStyle-Width="30px"
                 DataField="ID" ReadOnly="True" SortExpression="ID" />
<asp:TemplateColumn HeaderText="Name" SortExpression="Name"
                    ItemStyle-Width="150px"
   <ItemTemplate>
      <asp:Label runat="server" Text='<%#
                DataBinder.Eval(Container.DataItem, "Name") %>' />
   </ItemTemplate>
   <EditItemTemplate>
      <asp:TextBox ID="EditListName" runat="server"
                   Text='<%# DataBinder.Eval(Container.DataItem, "Name") %>'
                   Class="TextBox"
                   Width="150px"
      />
      <asp:RequiredFieldValidator Id="ValidateEditName"
                   runat="server"
                   ControlToValidate="EditListName"
                   Display="dynamic">* Name is required
      </asp:RequiredFieldValidator>
   </EditItemTemplate>
</asp:TemplateColumn>
```

```
<asp:TemplateColumn HeaderText="Description"
                    SortExpression="Description">
  <ItemTemplate>
    <asp:Label runat="server" Text='<%#
              DataBinder.Eval(Container.DataItem, "Description") %>' />
  </ItemTemplate>
  <EditItemTemplate>
    <asp:TextBox ID="EditListDescr" runat="server"
                 Text='<%# DataBinder.Eval(Container.DataItem,
                                           "Description") %>'
                 Class="TextBox"
                 Width="380px"
    />
  </EditItemTemplate>
</asp:TemplateColumn>
```

Первый столбец описывается в этой программе как обычный, не подлежащий редактированию столбец. Два других столбца позволяют продемонстрировать, каким образом по шаблону можно создавать столбец, использующий управляющий элемент `Label` для вывода значения или управляющий элемент `TextBox` для его редактирования. Соответствующее представление создается автоматически, когда мы входим в режим редактирования для данной строки или покидаем его. Обратите внимание на то, что текст в управляющем элементе `Label` и в управляющих элементах `TextBox` обрабатывается с помощью метода `DataBinder.Eval()`, который возвращает значение соответствующего столбца для считываемой строки, а для добавления нового списка требуется задать его имя.

Три остальных столбца представляют собой ссылки, которые позволяют переходить на страницы с дополнительной информацией:

```
<asp:HyperLinkColumn
            DataNavigateUrlField="ID"
            DataNavigateUrlFormatString="Subscribers.aspx?ListID={0}"
            DataTextFormatString="<img border=0 Alt='Show subscribers'
                                       src=./Images/imgAddresses.gif>"
            DataTextField="ID"
/>
<asp:HyperLinkColumn
            DataNavigateUrlField="ID"
            DataNavigateUrlFormatString="NewsHistory.aspx?ListID={0}"
            DataTextFormatString="<img border=0 Alt='Show past news for
                                       this list' src=./Images/imgMessage.gif>"
            DataTextField="ID"
/>
<asp:HyperLinkColumn
            DataNavigateUrlField="ID"
            DataNavigateUrlFormatString="SendEmail.aspx?ListID={0}"
            DataTextFormatString="<img border=0 Alt='Send a new mail
                                       for this list' src=./Images/imgMail.gif>"
            DataTextField="ID"
/>
```

Эти нередактируемые гиперссылки представляются в виде образов, а URL-адреса, на которые они указывают, создаются с помощью шаблона строки, описываемого в `DataNavigateUrlFormatString`: вместо маркера `{0}` вставляется `ID` соответствующей строки.

Класс BasePage

Прежде чем приступить к изучению файла `Lists.cs` с кодом поддержки, о котором упоминалось в предыдущем разделе, представим базовый класс, который будет использоваться всеми классами из файлов с кодом поддержки, используемыми

в этом примере. Этот класс называется `BasePage`, и он является производным от класса `System.Web.UI.Page`. Его описание сведено в следующую таблицу:

Класс BasePage Метод	Описание
`public string GetConnString()`	Извлекает строку соединения с базой данных из файла `web.config` — файла, в котором хранятся все установки конфигурации данного приложения
`public string GetSettingsFile()`	Извлекает имя файла, в котором хранятся все установки данного приложения, также находящегося в файле `web.config`

Этот класс обладает двумя общими методами, которые наследуются всеми производными классами. Это пример правильного использования объектно-ориентированного подхода, поскольку строка соединения и имя файла с установками могут понадобиться для всех этих классов, и наилучшим решением будет поместить функциональную возможность доступа к этой информации в одном классе, вместо того чтобы копировать и переносить ее каждый раз, когда у какого-либо класса возникает в ней необходимость. Обе функции реализованы весьма простым способом — они осуществляют поиск строки соединения и имени файла с установками в приведенном ниже разделе файла `web.config`, после чего помещают эту информацию в кэш-память:

```
<configuration>
   <appSettings>
      <add key="ConnString" value="Provider=Microsoft.Jet.OLEDB.4.0;
         Data Source=E:\Projects\NewsMailer\DB\NewsMailer.mdb" />

      <add key="SettingsFile" value="NewsMailer.Settings" />
   </appSettings>
</configuration>
```

Необходимо изменить вышеупомянутый ключ `ConnString` так, чтобы он правильно указывал на файл базы данных.

Код поддержки для Lists.aspx

После определения базового класса перейдем к рассмотрению того, что находится в файле `Lists.aspx` с кодом поддержки.

Класс Lists

В списке директив `using` этот класс находится в пространстве имен `NewsMailer` и является производным от `BasePage`:

```
namespace NewsMailer
{
   public class Lists : NewsMailer.BasePage
   {
```

После объявления нескольких защищенных управляющих элементов Web Form, мы подходим к тому месту, где реализованы функциональные возможности страницы. Ниже приводится обработчик, который запускается при загрузке страницы:

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!Page.IsPostBack)
   {
```

```
            // связать управляющие элементы страницы
            BindGrid();
        }
    }
```

В программе опционно вызывается метод `BindGrid()`, который наполняет таблицу списками рассылки:

```
protected void BindGrid()
{
    NewsMailer.ListDB lists = new NewsMailer.ListDB(GetConnSrting());
    // запросить все списки
    DataView myDV = lists.GetLists().Tables[0].DefaultView;

    // произвести сортировку данных в соответствии со значением SortExpression
    if ( ListsGrid.Attributes["SortExpression"] != null )
        myDV.Sort = ListGrid.Attributes["SortExpression"];

    ListGrid.DataSource = myDV;
    ListGrid.DataBind();
}
```

Эта функция вызывается, только если свойство `IsPostBack` данной страницы имеет значение `false`, т. е. когда страница либо загружается впервые, либо обновляется пользователем с помощью команды браузера `Refresh/Reload`, но не при повторной загрузке страницы, поскольку пользователь уже отправил некоторые данные форме. На самом деле, после того как страница отправляется обратно, состояние управляющих элементов сохраняется в скрытом управляющем HTML-элементе, а затем автоматически восстанавливается, что позволяет избежать отправки повторного запроса к базе данных.

`BindGrid()` — это, вероятно, наиболее важная функция любой страницы, поскольку именно с ее помощью страница получает данные из базы данных и выводит полученные записи в табличном представлении. В первой строке этой функции создается экземпляр объекта `ListsDB` с помощью библиотеки `NewsMailerDB`: ее конструктор принимает строку соединения, используемую всеми ее методами, а сама строка соединения возвращается методом `GetConnString()` класса `BasePage`:

```
NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
```

В следующей строке создается набор данных, и выводимые данные единственной таблицы сохраняются в частной переменной:

```
DataView myDV = lists.GetLists().Tables[0].DefaultView;
```

Выводимые данные затем сортируются в соответствии с атрибутом табличного представления, который представляет собой атрибут, задаваемый пользователем: в момент первого вызова функции этот атрибут имеет значение `null`, а выводимые данные вообще не подвергаются сортировке. В последних двух строках осуществляется привязка выводимых данных к свойству `DataSource` и вызывается метод `DataBind()` для вывода строк.

Сортировка списков

Когда происходит щелчок мышью на заголовке столбца, то вызывается функция `ListsGrid_Sort()`, имеющая следующий вид:

```
protected void ListsGrid_Sort(Object sender, DataGridSortCommandEventArgs e)
{
    ShowAddNewControls(false);
    ListsGrid.EditItemIndex = -1;
```

```
    // установить атрибут SortExpression, который будет использоваться
    // для сортировки данных в методе BindGrid
    ListsGrid.Attributes["SortExpression"]=e.SortExpression.ToString();
    BindGrid();
}
```

Прежде всего эта функция прячет управляющие элементы для включения нового списка, если они видимы в данный момент. Затем она отменяет режим редактирования строки, если он был установлен. Наконец, она получает выражение сортировки, по которому будет осуществляться фактическая сортировка, и название столбца, а затем обновляет содержимое управляющего элемента табличного представления данных, вызывая метод `BindGrid()`.

Редактирование и обновление списков

Мы переходим к реализации методов, которые осуществляют редактирование и обновление списков:

```
protected void ListsGrid_Edit(object sender, DataGridCommandEventArgs e)
{
    ShowAddNewControls(false);
    // начать редактирование
    ListsGrid.EditItemIndex = (int)e.Item.ItemIndex;
    BindGrid();
}

protected void ListsGrid_CancelEdit(object sender, DataGridCommandEventArgs
{
    ListsGrid.EditItemIndex = -1;
    BindGrid();
}

protected void ListsGrid_Update(object sender, DataGridCommandEventArgs e)
{
    if (Page.IsValid)
    {
        // получить новые данные из диалоговых окон
        string ListName = ((TextBox)e.Item.FindControl("EditListName")).Text;
        string ListDescr = ((TextBox)e.Item.FindControl("EditListDescr")).Text;
        int ListID = (int)ListsGrid.DataKeys[e.Item.ItemIndex];

        NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
        // обновить значения
        lists.Update(ListID, ListName, ListDescr);

        ListsGrid.EditItemIndex = -1;
        BindGrid();
    }
}
```

Первая процедура `ListGrid_Edit()`, вызываемая при наступления события, прячет управляющие элементы для включения новой строки и присваивает свойству `EditItemIndex` в качестве значения индекс редактируемой строки; это значение извлекается из объекта `Item`, который представляется в качестве свойства объекта `e`, передаваемого функции в качестве параметра. Наконец, опять вызывается метод `BindGrid()` для привязки данных текущей строки к двум текстовым окнам. Когда нажимается иконка X, свойству `EditItemIndex` присваивается −1 и таблица обновляется.

Наибольший интерес представляет то, каким образом происходит обновление строки, осуществляемое процедурой `ListGrid_Update()`. Выполнение кода происходит, только если страница является допустимой, т. е. когда соблюдены все правила проверки допустимости управляющих элементов. В этом случае текстовое окно `Name` должно содержать допустимое имя. При успешной проверке выполнение функции продолжается, в результате чего извлекаются текущие значения из обоих текстовых окон. Ссылка на каждое из окон возвращается методом `FindControl()` объекта `Item`, после чего их содержимое считывается посредством свойства `Text`. Для считывания текста из управляющего элемента `TextBox` необходимо текстовое окно, которое не возвращается методом `FindControl()` в явном виде, поэтому приходится вручную приводить возвращаемый объект к типу `TextBox`. Последний необходимый параметр — это идентификатор обновляемой строки, получить который не составляет никакого труда, поскольку данная колонка была объявлена в качестве первичного ключа таблицы данных, поэтому вся работа сводится к считыванию значения свойства. В заключение создается объект `ListsDB` и вызывается метод `Update()`, которому передается идентификатор списка, а также новые имя и описание; в результате этого изменения вносятся в базу данных, и остается только обновить таблицу, чтобы эти изменения нашли в ней свое отражение.

Удаление списка

Процедура обработки события `ListGrid_Delete()` обладает в точности такой же структурой, что и только что рассмотренные события, хотя она несколько проще; единственный параметр, который необходимо передавать методу `Delete()` объекта `ListsDB`, — это идентификатор списка, возвращаемый перечислимым свойством `DataKey`:

```
protected void ListsGrid_Delete(object sender, DataGridCommandEventArgs e)
{
   ShowAddControls(false);
   ListsGrid.EditItemIndex = -1;
   NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
   // получить идентификатор списка и удалить его
   Lists.Delete( (int)ListsGrid.DataKeys[e.Item.ItemIndex] );
   BindGrid();
}
```

Создание списка

Последнее, с чем нам осталось познакомиться, — это добавление новой строки. При нажатии кнопки Create на экран выводятся два текстовых окна и соответствующие иконки для подтверждения или отказа от выполнения операции, в то время как сама кнопка Create становится невидимой. Кроме того, свойству таблицы данных `EditItemIndex` присваивается значение –1, что предотвращает выполнение каких бы то ни было операций редактирования. Ниже приводится код для обработки этого события, а также код функции, которая выводит или прячет управляющие элементы в зависимости от значения передаваемого ей логического параметра:

```
protected void Create_Click(object sender, EventArgs e)
{
   // показать диалоговые окна и кнопки для добавления новой записи
   ShowAddNewControls(true);
   ListsGrid.EditItemIndex = -1;
   BindGrid();
}
```

```
protected void ShowAddNewControls(bool ShowControls)
{
   // показать / спрятать управляющие элементы для добавления новой записи
   NewListName.Text=" ";
   NewListDescr.Text=" ";
   AddNewControlsRow.Visible = !ShowControls;
   CreateNewRow.Visible=!ShowControls;
}
```

Иконка, добавляющая новый список, представляет собой не что иное, как кнопку, для которой обработка события "щелчок мышью" имеет следующий вид:

```
protected void AddNew_Click(object sender, EventArgs e)
{
   if (Page.IsValid)
   {
      NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
      // добавить новую запись
      lists.Add(NewListName.Text, NewListDescr.Text);
      ShowAddNewControls(false);
      BindGrid();
   }
}
}
```

Если страница является допустимой, как в случае события `Update()`, то программа получает содержимое двух текстовых окон и передает эти строки методу `Add()` объекта `ListsDB`. В данном случае код оказывается несколько проще, поскольку можно непосредственно ссылаться на оба окна, так как других управляющих элементов с такими именами не существует.

Иконка, отменяющая операцию добавления, представляет собой управляющий элемент `ButtonLink`, который, будучи нажатым, прячет все управляющие элементы для имени и описания нового списка и восстанавливает все в том виде, в каком это выводилось по умолчанию. Ниже приводится код для обработки события `Click`:

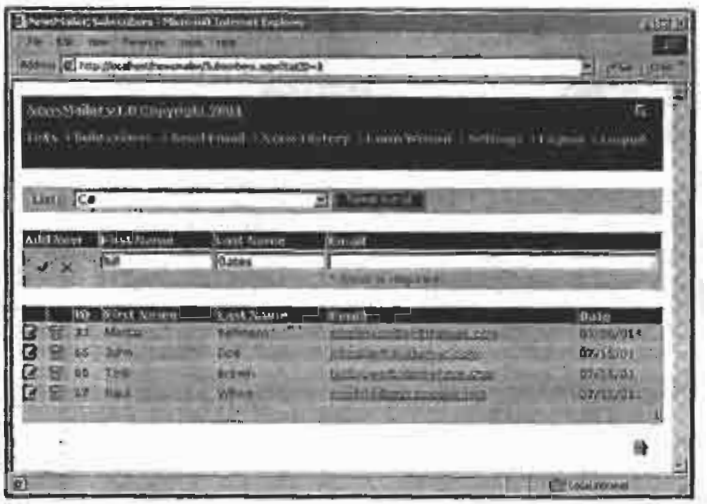
```
protected void CancelAddNew_Click(object sender, EventArgs e)
{
   ShowAddNewControls(false);
}
```

Здесь следует обратить внимание на один очень важный момент. Нажатие кнопки приводит к тому, что страница отправляется обратно. В этот момент производится проверка допустимости управляющих элементов с именем и описанием, и, если имя нового списка отсутствует, отправка страницы будет прекращена. Чтобы избавиться от этой проблемы, можно установить режим, при котором нажатие этой кнопки не приводит к проверке допустимости формы, присвоив ее свойству `CuasesValidtion` значение `false`.

На этом код для данной страницы завершен. Для создания страницы потребовалось не такое уж большое количество кода, особенно по сравнению с кодом, который был бы необходим при использовании традиционного ASP — даже простое создание многофункционального табличного представления данных потребовало бы несколько сотен (если не тысяч) строк сложного кода на ASP.

Управление подписчиками

Страница для управления подписчиками списков очень похожа на только что проанализированную страницу, поэтому в данном разделе мы остановимся только на ее отличиях. На рисунке (см. стр. 823) страница изображена в том виде, в каком



она выводится в процессе добавления нового подписчика. Пользователь произвел попытку подписаться, не указав своего адреса электронной почты, поэтому программа проверки допустимости сообщает ему, что поле Email должно быть заполнено обязательно.

Страница Subscribers.aspx

Как видно на рисунке, эта страница отличается от предыдущей. Одно из отличий заключается в том, что в таблице присутствует другое количество столбцов и, кроме того, имеется раскрывающееся окно, в где можно осуществлять выбор списка рассылки и просматривать подписчиков этого списка. Помимо этого есть кнопка, которая переносит пользователя на страницу, позволяющую отправить новое сообщение по данному списку рассылки.

Что касается столбцов, то все отличия сводятся к добавлению нескольких новых столбцов с шаблонами, к изменению имени столбца при вызове метода `DataBinder.Eval()`, а также к измененню определенных свойств некоторых столбцов без шаблонов. Код, относящийся к этой странице, оказывается довольно длинным. Ниже приводится его первая часть, в которой описывается комбинированное окно и кнопка **Send Email** (отправка электронной почты):

```
<asp:table id="TableLists" runat="server" CssClass="Grid_General>
   <asp:TableRow>
      <asp:TableCell width="50px"
         HorizontalAlign="Center"><b>List:</b></asp:TableCell>
      <asp:TableCell width="280px"
<asp:DropDownList runat="server" ID="ListDropDown"
            DataTextField="Name" DataValueField="ID" width="280px"
            Class="TextBox" AutoPostBack="True"
            OnSelectedIndexChanger="ListsDropDown_IndexChanged" />
      </asp:TableCell>
      <asp:TableCell>
         <asp:Button runat="server" Text="Send Email"
            ID="SendEmail" CssClass="Button"
            OnClick="SendEmail_Click" Width="100px" />
      </asp:TableCell>
   <asp:TableRow>
<asp:table>
```

Необходимо обратить внимание на то, что свойству раскрывающегося окна `AutoPostBack` присвоено значение `true`. Это приводит к тому, что страница будет отправляться обратно каждый раз, когда пользователь изменяет выбранный элемент. В свою очередь, это влечет за собой наступление события `OnSelectedIndexChanged` и, соответственно, вызов процедуры `ListsDropDown_IndexChanged()`.

Остальная часть страницы аналогична тому, что мы видели раньше. Рассмотрим дополнительные строки, которые были добавлены к объявлению таблицы данных:

```
<asp:DataGrid id="SubscrGrid" runat="server"
   AllowPaging="True"
   PageSize="20"
```

```
    PagerStyle-HorizontalAlign="Right"
    PagerStyle-PageButtonCount="20"
    PagerStyle-Mode="NumericPages"
    OnPageIndexChanged="SubscrGrid_PageChanged"
...
```

В этих строках соответственно определяется, что данные будут выводиться в виде страниц по 20 строк и что пользователь будет иметь возможность осуществлять поиск и перемещение по номерам страниц, расположенных внизу справа, а также задается процедура, которая будет обрабатывать события, связанные с изменением страницы.

Код поддержки для Subscribers.aspx

Что касается кода на ASP.NET, то код поддержки для `Subscribers.aspx` не очень отличается от страницы `Lists`. В нем появилось несколько строк для работы с вновь добавленными управляющими элементами. Этот код выполняется, когда происходит загрузка страницы:

```
protected void Page_Load(object sender, EventArgs e)
{
   // получить ListID из QueryString
   string ListID =Request.Params["ListID"];
   SubscrGrid.Attributes["ListID"] = ListID;

   If (!Page.IsPostBack)
   {
      // загрузить все доступные списки в управляющий элемент DropDown
      NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
      DataView myDV = lists.GetLists().Tables[0].DefaultView;
      ListsDropDown.DataSource = myDV;
      ListsDropDown.DataBind();

      If (ListID!=null)
      {
         // выбрать элемент DropDown в соответствии с ListID
         forearch (ListItem item in ListsDropDown.Items)
            if (item.Value==ListID)
               ListsDropDown.SelectedIndex =
                  ListsDropDown.Items.IndexOf(item);
      }

      // упорядочить управляющие элементы на странице
      BindGrid();
   }
}
```

С правой стороны каждой строки таблицы на страннице `Lists.aspx` располагалась иконка, которая, при щелчке на ней мышью, переносила пользователя на страницу `Subscribers.aspx`, чтобы он мог просмотреть всех подписчиков данного списка. Для этого использовалась ссылка на страницу `Subscribers.aspx`, а в строку запроса в качестве параметра добавлялся идентификатор списка. Далее это значение извлекается из строки запроса и сохраняется в качестве атрибута элемента, управляющего табличным представлением данных, так как оно будет использоваться в тот момент, когда в этом коде будет вызываться метод `GetSubscribers()` для данного списка.

Двигаясь далее, мы обнаружим, что в дополнение к обычному вызову метода `BindGrid()` на этот раз используется раскрывающийся список, который наполняется имеющимися в наличии списками рассылки. Это осуществляется непосредственно

внутри данной процедуры посредством обращения к методу `GetLists()` класса `ListsDB` и привязки получаемой в результате таблицы в качестве источника данных для управляющего элемента. Вновь добавленные элементы считываются один за другим и выбирается элемент, идентификатор которого совпадает с идентификатором, хранящимся в строке запроса.

В метод `BindGrid()` для этой страницы включен некоторый дополнительный код, считывающий идентификатор списка, для которого требуется извлечь подписчиков, и, кроме того, в ней для получения требуемого набора данных (то есть таблицы) используется класс `SubscribersDB`:

```
protected void BindGrid()
{
   int ListID;

   // получить значение ListID из атрибутов Grid
   if ( SubscrGrid.Attributes["ListID"] != null )
      ListID = int. Parse(SubscrGrid.Attributes["ListID"]);
   else
      ListID = int.Parse(ListsDropDown.SelectedItem.Value);

   // получить всех подписчиков данного листа
   NewsMailer.SubscribersDB subscribers =
                        new NewsMailer.SubscribersDB(GetConnString());
   DataView myDV = subscribers.GetSubscribers(
      ListID).Tables[0].DefaultView;

   // отсортировать все записи в соответствии с
   // атрибутом SortExpression DataGris
   if ( SubscrGrid.Attributes["SortExpression"]!=" ";
      myDV.Sort = SubscrGrid.Attributes["SortExpression"];

   SubscrGrid.DataSource = myDV;
   SubscrGrid.DataBind();
}
```

Эта функция отвечает за выбор элемента в раскрывающемся списке:

```
protected void ListsDropDown_IndexChanged(object sender, EventArgs e)
{
   // загрузить страницу, чтобы показать записи выбранного листа
   Response.Redirect("Subscribers.aspx?ListID=" +
      ListsDropDown.SelectedItem.Value);
}
```

В результате появляется возможность повторно направить браузер на ту же самую страницу, но на этот раз с идентификатором выбранного элемента, что позволит произвести загрузку соответствующих подписчиков при очередном обращении к методу `BindGrid()`.

Когда пользователь двигается по другим страницам таблицы данных, необходимо запретить переход в режим редактирования. Это означает, что, пока не будет произведена повторная привязка таблицы данных, необходимо спрятать элементы, управляющие добавлением новой строки, а также не выводить сообщение об ошибке, которая возникает, если мы пытаемся добавить уже существующего в списке подписчика. Необходимо присвоить свойству `CurrentPageIndex` фактическое значение, для того чтобы внести изменения в текущую страницу:

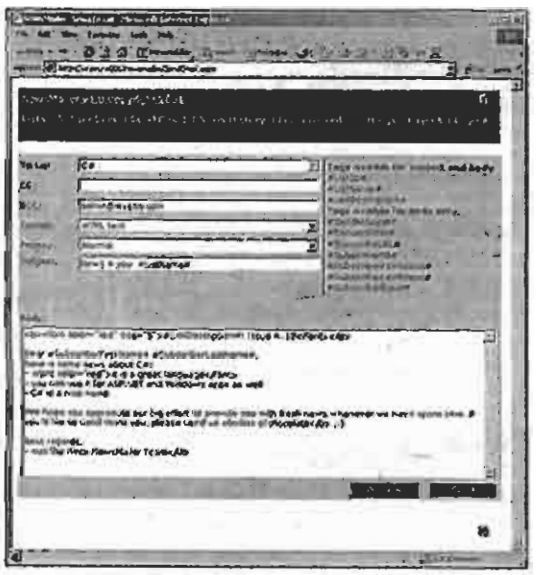
```
protected void SubscrGrid_PageChanged(object sender,
   DataGridPageChangedEventArgs e)
{
   SubscrGrid.EditItemIndex = -1
   AddError.Visible = false;
```

```
        ShowAddNewControls(false);
        // изменить текущую страницу для чтения данных
        SubscrGrid.CurrentPageIndex = e.NewPageIndex;
        BindGrid();
    }
```

В заключение процедура, которая обрабатывает событие, наступающее при нажатии кнопки **Send Email**, передает управление на страницу `SendEmail.aspx` с идентификатором текущего списка в качестве параметра — точно так же, как это делалось для раскрывающихся списков:

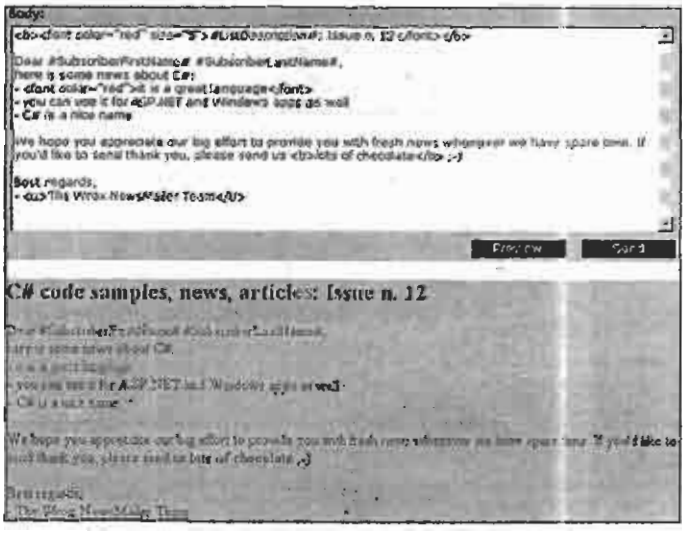
```
protected void SendEmail_Click(object sender, EventArgs e)
{
    // вернуться к странице, чтобы отослать Email,
    // добавить значение ListID к QueryString
    Response.Redirect("SendEmail.aspx?ListID=" +
        ListDropDown.SelectedItem.Value);
}
```

Отправка электронной почты подписчикам



Мы достигли критической точки в реализации приложения — теперь требуется разработать и написать код для страницы, которая позволяла бы администратору составлять сообщение, рассылать его всем подписчикам из списка и заносить его в архив в БД для последующего просмотра. Рисунок слева дает представление о том, как это все должно функционировать.

С помощью этой страницы администратор получает возможность создать само сообщение, отправляемое по электронной почте, и задать практически все опции, касающиеся отправки сообщений, которые являются общими для большинства клиентов: CC, BCC, формат электронного сообщения (простой текст или HTML), приоритет, предмет сообщения и его тело. Кроме того, можно использовать несколько отформатированных специальным образом строк (располагающихся с правой стороны), которые будут автоматически заменяться на соответствующие им значения для каждого сообщения, отправляемого по электронной почте посредством используещейся по умолчанию SMTP-службы, которая инсталлируется вместе с Windows 2000 Server, поэтому не потребуется ни каких-либо дополнительных средств, ни каких-либо серверов сторонних производителей для отправки сообщений. Кроме того, можно предварительно просмотреть сообщение, хотя при этом строки, ограниченные символами "#", не транслируются, поскольку предварительный просмотр не является специфическим для какого-либо конкретного подписчика. На рисунке справа изображено предварительное представление тела сообщения с предыдущего рисунка.



Страница SendEmail.aspx

Страница ASP.NET содержит код, аналогичный коду, встречавшемуся ранее, здесь он не будем воспроизводиться. Управляющие элементы в виде текстовых окон, уникальные для данной страницы, используются для вывода значений свойств `NewsSubject` и `Signature`, которые хранятся в XML-файле с установками и которые извлекаются кодом поддержки и передаются странице.

Код поддержки для SendEamail.aspx

Табличная сетка для вывода данных не используется на этой странице, поэтому данный файл с кодом поддержки существенно отличается от тех двух файлов с которыми мы сталкивались перед этим. Рассмотрим код метода `Page_Load()`, в котором части программы, отличающиеся от предшествующих файлов, выделены серым цветом:

```
// Используются две переменные для хранения конфигурационных установок
public string Signature_Value;
public string Subject_Value;

protected void Page_Load(object sender, EventArgs e)
{
   string ListID = Request.Params["ListID"]

   if (!Page.IsPostBack)
   {
      // загрузить все списки, перечисленные в управляющем элементе DropDown
      NewsMailer.ListsDB lists = new NewsMailer.ListsDB(GetConnString());
      DataView myDV = lists.GetLists().Tables[0].DefaultView;
      ListsDropDown.DataSource = myDV;

      // загрузить подписи и новые объекты из базы данных
      NewsMailer.Appsettings settings =
         NewsMailer.AppConfig.GetSettings(Server.MapPath(GetSettingsFile()));
      Signature_Value = settings.Signature;
      Subject_Value = settings.NewsSubject;
      Page.DataBind();

      if (ListID!=null)
      {
         foreach (ListItem item in ListsDropDown.Items)
            if (item.Value==ListID)
               ListsDropDown.SelectedIndex =
                  ListsDropDown.Items.IndexOf(item);
      }
   }
}
```

Модуль `NewsMailerConfig` используется для получения всех установок из XML-файла, а сигнатура и предмет новостей сохраняются в общих переменных.

Заметьте, что имя файла, в котором хранятся установки, извлекается либо из файла `web.config`, либо из кэш-памяти посредством функции `GetSettingsFile()` класса `BaseClass`, а возвращенное значение преобразуется в реальный путь с помощью вызова метода `Server.MapPath()`, который позволяет получить полный путь к файлу.

Когда выполняется метод `DataBind()`,осуществляющий привязку всех дочерних управляющих элементов текущей формы к даиным, то происходит привязка к значениям двух переменных — `Signature_Value` и `Subject_Value` — и их вывод в текстовых окнах.

Ниже приводится код для кнопки Preview:

```
protected void Preview_Click(object sender, EventArgs e)
{
   string MsgText = Body.Text;
   int ListID = int.Parse(ListsDropDown.SelectedItem.Value);

   // заменить теги по спискам и установкам на их новые значения
   MsgText = NewsMailer.Utilities.ProcessListTags(MsgText,
                                                  GetConnString(), ListID);
   MsgText = NewsMailer.Utilities.ProcessListTags(MsgText, GetConnString());

   // осуществить кодировку тела, если это не формат HTML
   if (FormatDropDown.SelectedIndex==0)
      MsgText = HttpUtility.HtmlEncode(MsgText);
   // заменить новые линии символами <br>
   MsgText = MsgText.Replace("\n", "<br>");
   // убрать символы возврата каретки
   MsgText = MsgText.Replace("\r", "");

   MsgPreview.Text = MsgText;
   MsgPreview.Visible = true;
}
```

В этом коде происходит вызов некоторых функций из класса Utilities, которые осуществляют замены определенных тегов на соответствующие им значения, кодирование текста в том случае, если в качестве формата выбран обыкновенный текст (например, тег <b> заменяется на <b>), заменяют все переходы к новой строке на тег
 и удаляют символы "возврат каретки", поскольку символы '\n' и '\r' не распознаются в HTML. В завершение тело сообщения выводится в управляющем элементе предварительного просмотра.

Перейдем к коду для кнопки Send Email:

```
protected void Send_Click (object sender, EventArgs e)
{
   Server.ScriptTimeout = 3600;

   Strins MsgBody = Body.Text;
   Int ListID = int.Parse(ListDropDown.SelectedItem.Value);
   String MsgSubject = NewsMailer.Utilties.ProcessListTags(
                                    Subject.Text, GetConnString(), ListID);

   NewsMailer.AppSettings =
      NewsMailer.AppConfig.GetSettings(Server.MapPath(GetSettingsFile()));

   NewsMailer.NewsHistoryDB news = new NewsMailer. NewsHistoryDB(
                                                      GetConnString());

   // заменить теги по спискам и установкам на их новые значения
   MsgBody = NewsMailer.Utilities.ProcessListTags(MsgBody,
                                                  GetConnString(), ListID);
   MsgBody = NewsMailer.Utilities.ProcessSettingsTags(MsgBody,
                                                    GetConnString());

   // добавить новую запись в архив новостей
   news.Add(ListID, MsgSubject,MsgBody, (FormatDropDown.SelectedIndex==1));

   // создать новое почтовое сообщение
   MailMessage MailMsg = new MailMessage();
   MailMsg.From = settings.SenderName + " <" + settings.SenderEmail + ">";
   MailMsg.Cc = CC.Text;
   MailMsg.Bcc = BCC.Text;
```

```
    MailMsg.Priority = (MailPriority)PriorityDropDown.SelectedIndex;
    MailMsg.BodyFormat = (MailFormat)FormatDropDown.SelectedIndex;
    MailMsg.Subject = MsgSubject;

    NewsMailer.SubscribersDB subscribers = new NewsMailer.SubscribersDB(
                                                        GetConnString());

    // получить всех подписчиков данного листа
    DataTable dt = subscribers.GetSubscribers(ListDB).Tables[0];
    foreach (DataRow dr in dt.Rows)
    {
       // заменить теги подписчика на их новые значения
       MailMsg.To = (srting)dr["Email"];
       MailMsg.Body = NewsMailer.Utilities.ProcessSubscriberTags(MsgBody,
                                          GetConnString(), (int)dr["ID"]);
       // послать это сообщение данному подписчику
       SmtpMail.Send(MailMsg );
    }

    // показать количество отосланных сообщений
    MsgPreview.Text = string.Format(
                        "<center><b>{0} mails have been sent</b></center>",
                        dt.Rows.Count.ToString());
    MsgPreview.Visible = true;
}
```

Эта программа более длинная, но работает она аналогично `Preview_Click`. Она выполняет следующие шаги:

- Специальные теги, находящиеся в списке, и общие установки заменяются на соответствующие им значения, а при необходимости производится кодирование текста
- Сообщение добавляется в архив с помощью метода `Add()` класса `NewsHistory`
- Создается объект `MailMessage` и присваиваются значения его свойствам, общим для всех подписчиков
- Для получения всех подписчиков текущего списка используется экземпляр класса `SubscribersDB`
- Для каждого подписчика, полученного в результате запроса, производится установка его адреса электронной почты в качестве адреса назначения для отправляемого объекта, после чего происходит отправка почты
- В управляющем элементе предварительного просмотра выводится число отправленных сообщений

В рассмотренном коде существует несколько моментов, на которые следует обратить внимание. Прежде всего в дополнение к обычным пространствам имен, которые объявлялись и для других страниц, на этот раз необходимо объявлять еще и пространство имен `System.Web.Mail`, содержащее в себе классы, позволяющие создавать сообщения и отправлять их по электронной почте посредством использующейся по умолчанию SMTP-службы, инсталлируемой на Windows 2000 Server.

Следующий момент, на который следует обратить внимание, имеет отношение к самой первой строке кода:

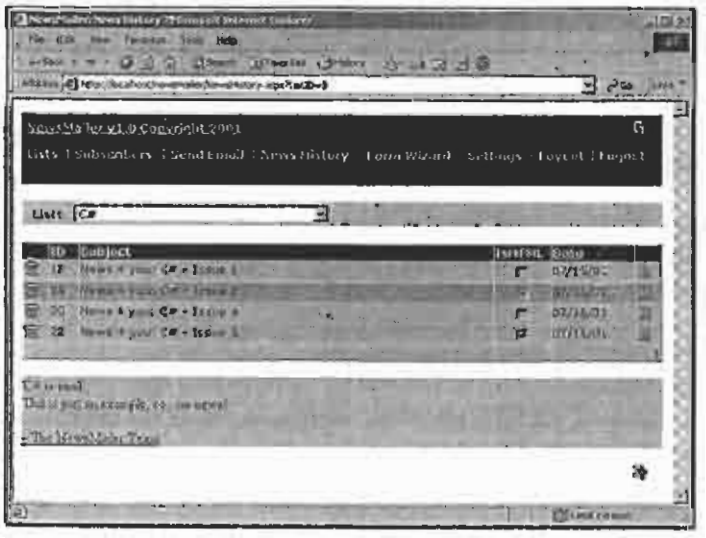
```
Server.ScriptTimeout = 3600;
```

Отправка отдельного сообщения — очень быстрое задание, однако если есть несколько тысяч подписчиков, то на это может потребоваться значительное время — несколько минут или даже больше. Те из вас, кто уже знаком с программированием на традиционном ASP, знают, что по умолчанию странице дается 90 секунд на выполнение кода сценария и на отправку всего сгенерированного HTML-кода браузеру. Это значение, использующееся по умолчанию, может быть изменено администратором IIS, либо "зашито" в программу страницы — что мы в данном случае и делаем, устанавливая максимальное время, в течение которого страница должна завершить свое выполнение, равное одному часу.

Класс Utilities

В файле `SendEmail.cs` с кодом поддержки для замены специальных тегов в теле и предмете сообщения используется класс `Utilities`. Ниже приводится исчерпывающий список методов класса. Обратите внимание на то, что они описаны как статические; это означает, что, для того чтобы ими воспользоваться, нет необходимости создавать экземпляр данного класса:

Класс Utilities Метод	Описание
`public static string ReadTextFile(string FilePath)`	Позволяет получать содержимое заданного файла
`public static bool SaveRextFile(string FilePath, string FileText)`	Сохраняет строку `FileText` в заданном файле и возвращает логическое значение, свидетельствующее об успешном или неуспешном завершении операции
`public static string ProcessListTags(string Message, string ConnString, int ListID)`	Заменяет теги списка, обрамленные символами "#", на значения, хранящиеся в базе данных
`public static string ProcessSubscriberTags(string Message, string ConnString, int SubscrID)`	Заменяет теги подписчика, обрамленные символами "#", на значения, хранящиеся в БД
`public static string ProcessSettingsTags(string Message, string SettingsFilePath)`	Заменяет теги установок, обрамленные символами "#", на значения, хранящиеся в БД



Работа с архивом новостей

Страница архива новостей выводит все предыдущие новости для всех списков рассылки либо в виде прямого текста, либо в виде HTML — в зависимости от того, в каком формате они отправлялись. На рисунке слева представлен внешний вид страницы архива.

Страница NewsHistory.aspx

Эта страница использует управляющий элемент табличного представления данных, но вместе с тем она обладает

некоторыми уникальными особенностями. Обратите внимание на то, что выбранный ряд выводится другим цветом, нежели все остальные, а у столбца `IsHTML` есть окно выбора, указывающее на то, должен ли вывод производится в виде HTML. Поле `Date` указывает на дату отправления данного сообщения. Поскольку большая часть кода не отличается от кода предыдущих страниц, мы не будем приводить весь текст полностью, а приведем только код, описывающий столбцы `IsHTML` и `Date`.

```
<Columns>
. . . Code removed for clarity

    <asp:BoundColumn HesderText="Subject" DataField="Subject"
      ReadOnly="True" SortExpression="Subject" />

    <asp:TemplateColumn HeaderText="IsHTML"
      ItemStyle-Width="60px" ItemStyle-HorizontalAlign="Center">
      <ItemTemplate>
        <asp:CheckBox runat="server" Enabled="false"
          Checked='<%#
          (bool)DataBinder.Eval(Container.DataItem, "IsHTML")
          %>'
        />
      </ItemTemplate>
    </asp:TemplateColumn>

    <asp:TemplateColumn HeaderText="Date"
      SortExpression="SentDate" ItemStyle-Width="90px">
      <ItemTemplate>
        <asp:Label runat="server"
          Text='<%# string.Format("{0:MM/dd/yy}",
            DateTime.Parse(DataBinder.Eval(Container.DataItem,
            "SentDate").ToString()) ) %>'
        />
      </ItemTemplate>
    </asp:TemplateColumn>
```

Как и для страницы Send Email, где мы привязываем общую переменную к текстовому окну, в данном случае столбец `IsHTML` может использоваться в режиме "только чтение", и в нем содержится отключенное окно выбора (отключенное, поскольку информация о прошлых сообщениях не может изменяться администратором посредством этой страницы), значение которого устанавливается в соответствии со значением поля `IsHTML` таблицы `NewsHistory`, хранящейся в базе данных, и приводится к типу `bool`.

Чтобы получить значения поля `Date`, используется метод `DateTimeParse()`, позволяющий преобразовать строку из базы данных в значение типа `DateTime`, которое затем снова преобразовывается в строку для вывода на экран.

Код поддержки для NewsHistory.aspx

Пояснения, которыми мы сопроводим данный код, будут очень короткими, поскольку большая часть операций, уже обсуждалась для других файлов с кодом поддержки. Единственной новой особенностью, заслуживающей обсуждения, является процедура, обрабатывающая событие таблицы данных `OnSelectedIndexChanged`. Когда некоторая строка оказывается выбранной, то тело соответствующего сообщения извлекается, перекодируется в обычный тест и выводится в управляющем элементе предварительного просмотра:

```
protected void NewsGrid_SelectionChanged(object sender, EventArgs e)
{
    string Body;
    NewsMailer.NewsHistoryDB news = new NewsMailer.NewsHisroryDB(GetConnString());
```

```
    // получить ID выбранной строки
    int NewsID = int.Parse(NewsGrid.DataKeys[NewsGrid.SelectedIndex].ToString());

    // получить массив новостей из базы данных
    Body = news. GetDetails(NewsID) ["Body].ToString();

    // если формат новостей не HTML, осуществить кодировку текста Body
    if (!Convert.ToBoolean(news.GetDetails(NewsID)["IsHTML"]))
       Body = HttpUtility.HtmlEncode(Body);

    // заменить новые строки символами <br>, чтобы показать их в элементе Label
    Body = Body.Replace("\n", ",br>");
    Body = Body.Replace("\"t", " ");

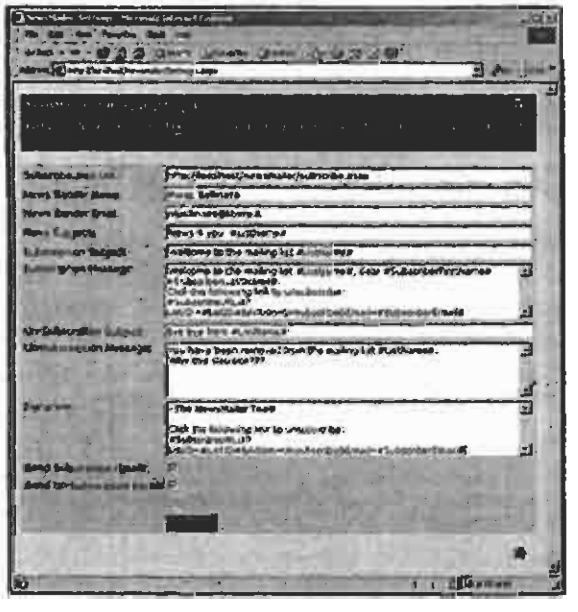
    // показать тело Body в элементе "предварительный просмотр"
    MsgPreview.Text = Body;
    Msg Preview.Visible = true;
}
```

Обратите внимание на то, что нам не приходится в данном случае заменять никаких специальных тегов, поскольку все они были заменены перед тем, как сообщение было помещено в архив, хранящийся в базе данных.

Кроме того, в тот момент, когда администратор изменяет просматриваемую страницу таблицы даиных, нам необходимо отказаться от выбраиной строки, поскольку эти действия автоматически не выполняются:

```
protected void NewsGrid_PageChanged(Object sender, DataGridPageEventArgs e)
{
   UnselectGridItem();
   // изменить текущую страницу для чтения данных
   NewsGrid.CurrentPageIndex = e.NewPageIndex;
   BindGrid();
}

protected void UnselectGridItem()
{
   // спрятать элемент "предварительный просмотр"
   NewsGrid.SelectedIndex = -1;
   MsgPreview.Text = " ";
   MsgPreview.Visible = false;
}
```



Изменение установок в режиме онлайн

Нам уже неоднократно приходилось использовать значения, хранящиеся в XML-файле с установками `NewsMailer.Settings`, которые мы определили в момент создания этого файла. Теперь нам потребуется web-страница, позволяющая модифицировать эти установки в режиме онлайн — без необходимости садиться перед web-сервером и вручную вносить изменения в базу данных. Страница, о которой идет речь, изображена на рисунке слева.

Страница Settings.aspx

Страница ASP.NET формируется с помощью набора управляющих элементов, к которым привязываются соответствующие установки, хранящиеся в базе данных `Settings`. Они привязываются к общим переменным, значения которых задаются в коде поддержки, как и для страницы Send Email. Настоящая страница является очень большой, для иллюстрации ее работы мы приводим участок кода, в котором описывается поле Sender Email, а также происходит наполнение соответствующего текстового окна с выполнением дополнительной проверки допустимости:

```
<asp:TableRow>
   <asp:TableCell Text="News Sender Email:" />
   <asp:TableCell>
      <asp:TextBox runat="servsr" Width="500px" ID="SenderEmail"
         Class="TextBox" Text='<%# SenderEmail_Value %>' />
      <asp:RequiredFieldValidator runat="server"
         ControlToValidate="SenderEmail"
         Display="dynamic"><br>* This field is required
      </asp:RequiredFieldValidator>
      <asp:RegularExpressionValidator runat="server"
            validationExpression=".*@.*\..*"
            ControlToValidate="SenderEmail"
            Display="dynamic"><br>* This Email address is not valid
      </asp:RegularExpressionValidator>
   </asp:TableCell>
</asp:TableRow>
```

Приведенный выше управляющий элемент является особенным, поскольку в нем присутствует не один, а два элемента, предназначенных для проверки допустимости: один из них позволяет гарантировать, что текстовое окно не является пустым в момент отправки формы, а второй — что строка с адресом электронной почты представлена в допустимом формате. В последнем случае используется `RegularExpressionValidator`, и, как следует из его названия, он выполняет проверку допустимости регулярного выражения. В настоящем примере используется следующее выражение:

```
<asp:RegularExpressionValidator runat="server"
   validationExpression=".*@.*\..*"
   ControlToValidate="SenderEmail"
   Display="dynamic"><br>* This Email address is not valid
</asp:RegularExpressionValidator>
```

Это означает, что строка должна начинаться с символов (.*), затем в ней должен находиться символ '@', потом произвольные символы, затем точка (для которой используется escape-последовательность \.) и, наконец, произвольные символы. Например, `mbellinaso@vb2themax.com` представляет собой допустимый адрес электронной почты, в то время как `mbellinaso@vb2themax` или `mbellinaso.libero.it` — недопустимые адреса.

Регулярные выражения представляют собой весьма мощный инструмент для проверки допустимости содержимого управляющего элемента, поскольку названные выражения могут быть детальными и сложными; их использованию посвящены целые книги. Кроме того, существует возможность при написании функции, производящей проверку допустнмости, использовать метод `CustomValidator`, который позволяет осуществлять еще более жесткий контроль.

Код поддержки для файла Settings.aspx

Код поддержки для внесения изменений в установки состоит из двух процедур: одна из них — это метод `Page_Load()`, предназначенный для извлечения установок из базы данных с помощью объектов `AppConfig` и `AppSettings` и привязки их к соответствующим серверным управляющим элементам посредством вызова метода `Page.DataBind()`:

```
protected void Page_Load(object sender,EventArgs e)
{
   if (!IsPostBack)
   {
      // загрузить все установки
      NewsMailer.AppSettings settings =
      NewsMailer.AppConfig GetSettings(Server.MapPath(GetSettingsFile()));

      SubscribeURL_Value = settings.SubscribeURL;
      SenderName_Value = settings.SenderName;
      SenderEmail_Value = settings.SenderEmail;
      NewsSubject_Value = settings.NewsSubject;
      SubscrSubject_Value = settings.SubscrSubject;
      SubscrMessage_Value = settings.SubscrMessage;
      UnSubscrSubject_Value = settings.UnSubscrSubject;
      UnSubscrMessage_Value = settings.UnSubscrMessage;
      Signature_Value = settings.Signature;
      SendSubscrEmail_Value = settings.SendSubscrEmail;
      SendUnSubscrEmail_Value = settings.SendUnSubscrEmail;

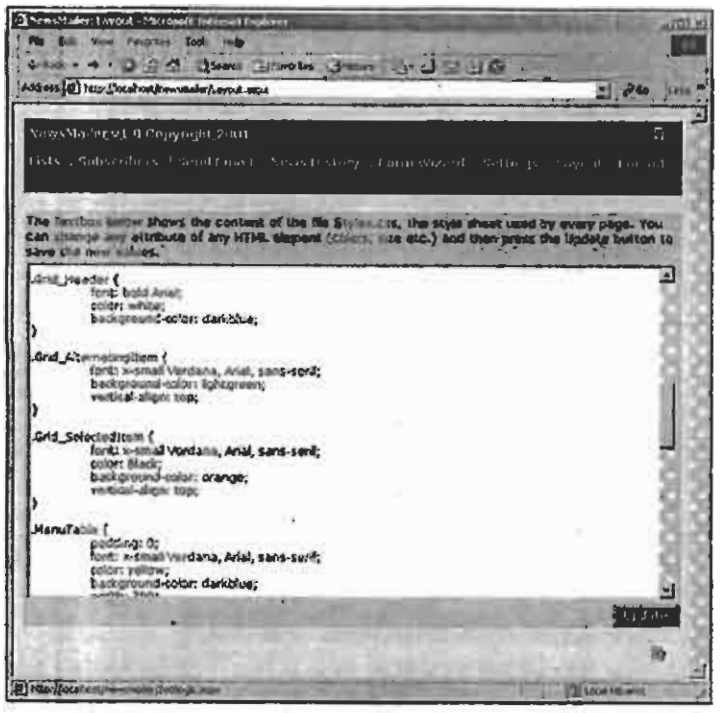
      Page.DataBind();
   }
}
```

Вторая процедура — это `Update_Click()`, которая вызывается в момент, когда мышь щелкает кнопку Update. Эта процедура собирает новые значения установок, находящиеся в управляющих элементах, присваивает их свойствам нового объекта `AppSettings` и вызывает метод `SaveSettings()` объекта `AppConfig` для сериализации объекта `settings` в XML-файл (в данном случае — в `NewsMailer.Settings`):

```
protected void Update_Click(object sender,EventArgs e)
{
   NewsMailer.AppSettings settings = new NewsMailer.AppSettings();
   // установить новые значения для всех свойств
   settings.SubscribeURL = SubscribeURL.Text;
   settings.SenderName = SenderName.Text ;
   settings.SenderEmail =SenderEmai.Text ;
   settings.NewsSubject = NewsSubject.Text;
   settings.SubscrSubject = SubscrSubject.Text;
   settings.SubscrMessage = SubscrMessage.Text;
   settings.UnSubscrSubject = UnSubscrSubject.Text;
   settings.UnSubscrMessage = UnSubscrMessage.Text;
   settings.Signature = Signature.Text ;
   settings.SendSubscrEmail = SendSubscrEmail.Checked;
   settings.SendUnSubscrEmail = SendUnSubscrEmail.Checked;
   NewsMailer.AppConfig.SaveSettings(Server.MapPath(GetSettingsFile()), settings);
}
```

Изменение внешнего представления в режиме онлайн

В самом начале этого раздела говорилось о том, что все стили для управляющих элементов хранятся в отдельной таблице стилей, а не внутри страницы. На самом деле, "зашитые" в программу цвета и другие параметры стилей внешнего



представления всегда оборачиваются головной болью, когда необходимо внести какие-либо изменения, так как приходится вручную проходить по всем страницам и изменять все вхождения для стиля, который мы хотим модифицировать. Наличие таблицы стилей позволяет существенно упростить жизнь. Однако, все равно было бы хорошо иметь возможность редактировать таблицу стилей в режиме онлайн, вместо того, чтобы вносить изменения в файл на локальном компьютере, а затем отправлять этот файл обратно на сервер. Реализация страницы, которая позволяет редактировать файлы, требует всего лишь нескольких минут и нескольких строк кода, и к созданию именно такой страницы мы сейчас и приступим.

На рисунке слева представлен наш простой редактор таблиц стилей.

Если вы знаете CSS, то вы получаете возможность изменять стили управляющих элементов ASP.NET, а полное изменение стиля всего пользовательского интерфейса — это вопрос нескольких секунд!

Страница Layout.aspx

Этот файл не очень отличается от других управляющих элементов ASP.NET, поэтому мы не будем приводить здесь весь код полностью. Изображенная ниже строка кода, однако, требует дальнейших пояснений:

```
<asp:Label ID="FileName" runat="Server" Text="Styles.css" Visible="false" />
```

В этой строке объявляется скрытая надпись, которая используется для задания имени файла, подлежащего редактированию; это означает, что, изменяя эту установку, мы получаем возможность редактировать произвольный текстовый файл. Администратору по-прежнему необходимо выполнить повторную загрузку, если он изменяет путь к файлу в скрытых управляющих элементах, однако это оказывается намного проще и быстрее, чем изменять значения в файлах с кодом поддержки, поскольку в таком случае приходнтся заново транслировать и загружать весь модуль. Всегда помните, что приложение должно распространяться в откомпилнрованном виде, а отсюда следует, что если какая-нибудь установка будет "зашита" в программу, то это приведет к тому, что администратор сайта, работающий удаленно, окажется не в состоянии изменить ее без помощи разработчика или без повторного компилирования исходного кода. Поэтому наилучшим решением является размещение установок в скрытых полях страницы ASP.NET, как показано в настоящем примере.

Код поддержки для страницы Layout.aspx

Код поддержки данной страницы считывает содержимое файла и привязывает его к текстовому окну в процедуре обработки событий `Page_Load()`, в то время как содержимое текстового окна считывается и сохраняется в указанном файле прн

обработке события `Update_Click()`. Обратите внимание на то, что для хранения текста файла используется переменная уровня класса `FileContent_Value`:

```
public string FileContent_Value;

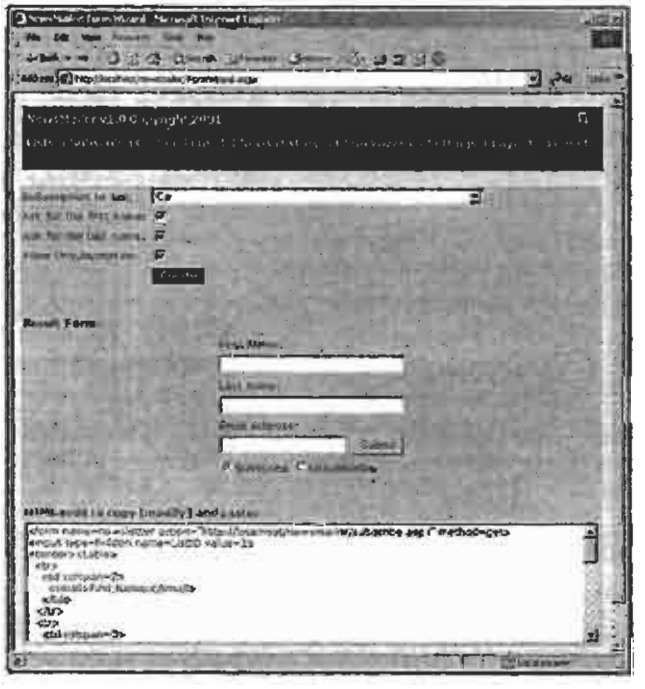
void Page_Load(object sender, EventArgs e)
{
   if (!Page.IsPostBack)
   {
      // прочитать содержимое выбранного файла
      FileContent_Value = NewsMailer.Utilities.ReadTextFile(
         Server.MapPath(FileName.Text));
      Page.DataBind();
   }
}

void Update_Click(object sender, EventArgs e)
{
   // сохранить файл с новым текстом
   NewsMailer.Utilities.ReadTextFile(
      Server.MapPath(FileName.Text), FileContent.Text);
}
```

Заметьте, что код, который фактически производит считывание содержимого текстового файла и его запись, находится в классе `Utilities`, который нами обсуждался ранее. Оба метода — `ReadTextFile()` и `SaveTextFile()` — описаны как статические, откуда следует, что нет необходимости создавать экземпляр класса `Utilities`, для того чтобы воспользоваться ими.

Создание подписной формы для пользователя

Все страницы, предназначенные для управления списками рассылки и подписчиками, отправки сообщений по электронной почте изменения внешнего представления, завершены. Осталось создать подписную форму, которая позволит конечному пользователю подписаться на некоторый список.



Вместо создания одной подписной формы, создадим страницу ASP.NET (см. рис. слева), которая будет автоматически генерировать HTML-форму для заданного списка рассылки.

Администратор сначала выбирает список рассылки, для которого он собирается создавать подписную форму, потом определяет необходимость наличия полей для первого и последнего имени (эти поля при подписке не являются обязательными, поэтому форма вполне может без них обойтись). Создаваемая форма будет иметь две кнопки, которые реализуют две возможности пользователя — подписаться и отказаться от подписки. При нажатии кнопки **Create** генерируется HTML-код, который помещается в расположенное в нижней части страницы текстовое окно, а сама форма выводится в центре страницы для предварительного просмотра. В этот момент у администратора есть возможность скопировать сгенерированный HTML и перенести его в произвольную страницу.

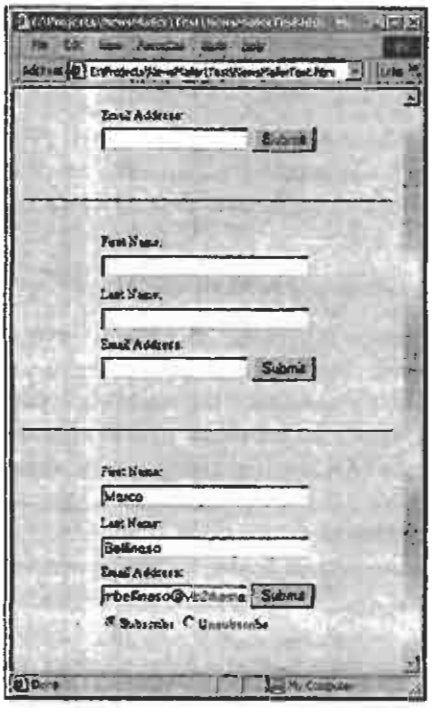
Страница FormWizard.aspx

Первая часть страницы, в которой находятся раскрывающийся список и текстовые окна, строится как обычно, с выравниванием всех управляющих элементов при помощи таблицы. Вторая часть таблицы, которая располагается на приведенном выше рисунке за надписью "Result Form:"(получеиная форма), по умолчанию является невидимой и выводится, только когда пользователь иажимает кнопку Create. Поскольку код этой страницы почти целиком состоит из управляющих элементов ASP.NET, с аналогами которых мы уже встречались ранее, этот код здесь ие приводится.

Код поддержки для страницы FormWizard.aspx

Код поддержки для этой страницы производит обращение к базе данных для заполнения раскрывающегося списка имеющимися в наличии списками рассылки, а также извлекает URL страннцы `Subscribers.aspx` из таблицы `Settings`. Остальная часть программы занимается созданием HTML-текста для формы с учетом выбранных опций. Обработчик событий `Create_Click()` создает HTML-строку, содержимое которой определяется в зависнмости от опций, выбранных администратором в верхней части формы. Поскольку в этом коде ие содержится чего-либо нового или интересиого, мы этот код воспроизводить не будем.

HTML-код, созданный обработчиком событий `Create_Click()`, выводится в управляющем элементе `Label` с именем `ResultForm`, что позволяет пользователю осуществлять предварительный просмотр результатов своей работы. Получаемый HTML не является закончеиным, поскольку в качестве корневого элемента в нем присутствует тег `<form>`, но при необходимости его можно легко скопировать из текстового окна в отдельный файл.



На рисунке слева можно увидеть некоторые тестовые формы, сгенерированные с использованием данной страницы.

Первая из представленных выше форм, была создана без использования всех трех возможных опций, вторая — с использованием двух первых опций (запрос первого и последнего имени), а третья форма создавалась с использованием всех трех имеющихся возможностей.

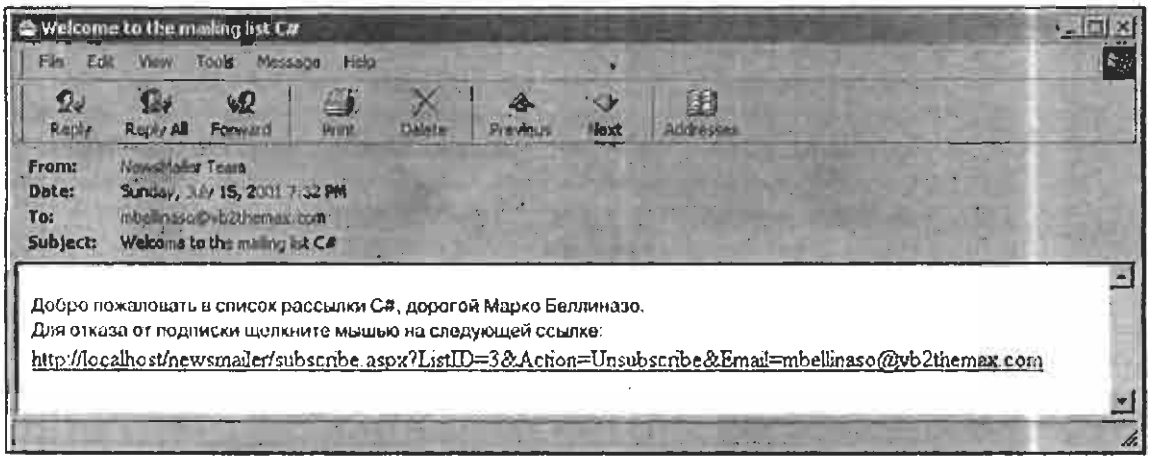
Совершенно очевидно, что, поскольку создаваемый код представляет собой обычный HTML, есть возможность видоизменять получаемый результат по своему усмотрению: например, изменить цвет, шрифт, ширину текстовых окон, использовать определенный рисунок для границ или фона или вывести комментарии, позволяющие пользователю получить дополнительную информацию относительно списка, на который он подписывается. Очень полезным представляется использование функции на JavaScript, которая позволяет перед фактической отправкой формы убедиться в том, что поле, содержащее адрес электронной почты, не является пустым.

Подписка на список рассылки

До этого момента мы смотрели на приложение с точки зрения администратора, а как быть с конечным пользователем? Посетитель зашел на наш web-сайт и решил подписаться на какой-либо список рассылки, но каким образом он может это

осуществить? Именно это мы сейчас и продемонстрируем. Страница с подписной формой может находится где угодно, даже на другом сервере, поскольку она указывает на полный URL страницы `Subscribe.aspx`.

Настала пора протестировать одну из подписных форм, представленных на странице, которая изображена на предшествующем рисунке и которая позволяет регистрировать новых подписчиков. Мы будем заполнять третью из форм, показанных на рисунке. Если на странице администратора выбрана установка Send Subscription Email (отправить сообщение о подписке), что происходит по умолчанию, если вы используете включенный в состав программного обеспечения файл с установками, то подписчик получит по электронной почте сообщение с подтверждением, содержащее информацию, заданную им самим:



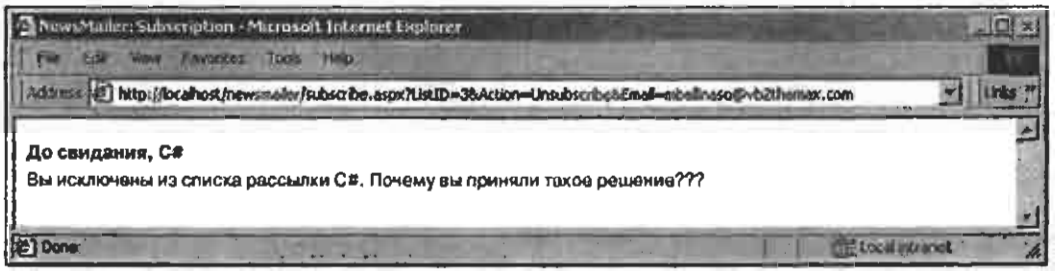
Последняя строка этого сообщения представляет собой ссылку, которая, в случае если ее нажать, удалит данного подписчика из списка рассылки. Эта ссылка создается динамически из приведенной ниже строки посредством замены тегов на соответствующие значения, извлеченные из XML-файла:

```
#SubscriberURL#?ListID=#ListID#&Action=Unsubscribe&Email=#SubscriberEmail#
```

Параметр `Action` отличает подписку от отказа от подписки. В данном случае ему присваивается значение `Unsubscribe`. Ссылка для осуществления подписки представляет собой точно такую же строку, но не содержащую параметра `Action`, или содержащую параметр `Action`, которому в качестве значения присваивается произвольная строка (только не `Unsubscribe`). Остальные параметры, входящие в ссылку,— это идентификатор списка рассылки (который хранится в форме в скрытом поле) и адрес электронной почты подписчика.

Сообщение о подтверждении посылается по электронной почте, только если есть соответствующая установка. Вы, однако, всегда будете получать web-страницу с подтверждением, в которой содержится тот же самый текст, что и в сообщении. Процедура отказа от подписки аналогична: администратор определяет, должно ли посылаться подтверждение, но на сайте в любом случае выводится соответствующая web-страница. Чтобы протестировать выполнение этой операции, откройте ссылку, присутствующую в предыдущем сообщении, это приведет к открытию браузера и выводу сообщения, представленного на рисунке на стр. 839.

Вы получите точно такой же результат, если выберете опцию `Unsubscribe` (отказ от подписки) в HTML-форме. У администратора есть также возможность добавлять или удалять подписчика с помощью страницы `Subscribers.aspx`.



Страница Subscribe.aspx

Код, предназначенный для создания этой страницы, является наиболее коротким из всех встречавшихся нам в этой главе:

```
<body>
   <asp:Label ID="Subject" runat="Server"
      CssClass="SubscrSubject" Visible="false" />
   <asp:Label ID="Message" runat="Server"
      CssClass="SubscrMessage" Visible="false" />
   <asp:Label runat="server" ID="EmailRequired" Visible="false"
      Text="The Email address is required" Cless=Error" />
</body>
```

На странице присутствует три надписи: для подтверждения, для тела сообщения и для сообщения об ошибке, если не задан адрес электроиной почты. Все эти надписи по умолчанию являются невидимыми. Они выводятся, только после того как пользователь выполняет операцию подписки или отказа от подписки и в соответствии с ее результатом.

Код поддержки для Subscribe.aspx

Весь код данного класса поддержки помещен в процедуру `Page_Load()`, поскольку все операции выполняются в момент загрузки страинцы н на ней не существует какнх-либо форм или управляющих элементов, которые приводят к наступлению других событий. Код для данного обработчика событий приводится ниже полностью:

```
protected void Page_Load(object sender,EventArgs e)
{
   int SubscrID;
   bool SendEmail;
   // считать ListID, FirstName, LastName, Email and Action
   // из QueryString, сгенерированной формой HTML
   int ListID = int.Parse(Request.Params["ListID"]);
   string FirstName = Request.Params["FirstName"] + " ";
   string LastName = Request.Params["LastName"] + " ";
   string Email = Request.Params["Email"] + " ";
   string Action = Request.Params["Action"] + " ";

   if (Email.Trim()=="")
   {
      // если объект Email не был определен,
      // вывести сообщение об ошибке и завершить работу
      EmailRequired.Visible = true;
   }
   else
   {
      string SubjectText;
      string MessageText;
```

```
// создать объекты для работы с установками и с подписчиком
NewsMailer.AppSettings settings =
   NewsMailer.AppConfig.GetSettings(
     . Server.MapPath(GetSettingsFile()));

NewsMailer.SubscribersDB subscribers = new
   NewsMailer.SubscribersDB(GetConnString());

// лишение подписки
if (Action.CompareTo("Unsubscribe")==0)
{
   // считать сообщение от подписчика
   // и проверить, требуется ли подтверждающий ответ по Email
   SendEmail = settings.SendUnSubscrEmail;
   SubjectText = settings.UnsubscrSubject;
   MessageText = settings.UnSubscrMessage;
   // считать ID этого подписчика из Email
   SubscrID = subscribers.GetSubscrID(ListID, Email);
   // заменить теги этого подписчика
   MessageText = NewsMailer.Utilities.ProcessSubscriberTags(
      MessageText, GetConnString(), SubscrID);
   // вычеркнуть подписчика
   subscribers.Delete(SubscrID);
}
else                              // оформить подписку
{
   // считать сообщение от подписчика
   // и проверить, требуется ли подтверждающий ответ по Email
   SendEmail = settings.SendSubscrEmail;
   SubjectText = settings.SubscrSubject;
   MessageText = settings.SubscrMessage;
   // добавить подписчика
   subscribers.Add(ListID, FirstName, LastName, Email);
   // получить ID только что зарегистрированного подписчика
   // или, если подписчик уже зарегистрирован
   // с известным Email адресом
   SubscrID = subscribers.GetSubscrID(ListID, Email);
   // заменить теги нового подписчика
   MessageText = NewsMailer.Utilities.ProcessSubscriberTags(
      MessageText, GetConnString(), SubscrID);
}

// заменить теги по установке и в списке
SubjectText = NewsMailer.Utilities.ProcessListTags(
   SubjectText, GetConnString(), ListID);
MessageText = NewsMailer.Utilities.ProcessListTags(
   MessageText, GetConnString(), ListID);
MessageText = NewsMailer.Utilities.ProcessSettingsTags(
   MessageText, GetConnString());

// если требуется подтверждающий ответ
if (SendEmail)
{
   // создание нового сообщения
   MailMessage MailMsg = new MailMessage();
   MailMsg.From = settings.SenderName + " <" +
      Settings.SenderEmail + ">";
   // заполнение поля "Кому" в Email
   MailMsg.To = Email;
   // заполнение поля "Содержание",
   // созданного для случая подтверждения
   MailMsg.Subject = SubjectText;
   MailMsg.Body = MessageText;
```

```
            // отсылка сообщения
            SmtpMail.Send(MailMsg );
        }

        // вывод сообщения, подтверждающего выполненное действие
        Subject.Text = SubjectText;
        Message.Text = MessageText;
        Subject.Visible = true;
        Message.Visible = true;
    }
}
```

В первых строках кода из строки запроса извлекаются идентификатор требуемого списка, имя подписчика и его адрес и переносятся в локальные переменные, с тем чтобы их было проще использовать в данной процедуре в дальнейшем. Прежде, чем переходить к выполнению дальнейших операций, программа выполняет проверку, не является ли параметр с адресом электронной почты пустым, и если является, то на экран выводится надпись об ошибке `EmailRequired` (адрес электронной почты необходим), а остальная часть процедуры не выполняется. Если адрес электронной почты присутствует, а параметр `Action` имеет значение `Unsubscribe`, то программа сначала задает предмет сообщения и текст подтверждения и заменяет специальные теги на информацию о конкретном подписчике, а затем удаляет информацию о данном подписчике из базы данных. В противном случае информация о подписчике сначала добавляется в базу данных, а затем происходит замена тегов. Замена тегов осуществляется после, поскольку методу `Utilities.ProcessSubscriberTags()` требуется передать в качестве параметра идентификатор подписчика, следовательно, информация о новом подписчике сначала должна быть занесена в БД. После этого производится замена специальных тегов, касающихся списка рассылки и установок, которая является необходимой независимо от того, подписывается ли пользователь или производит отказ от подписки. Если в файле установок хранится информация о том, что требуется отправить по электронной почте сообщение с подтверждением (состоящее из предмета сообщения и тела сообщения), то процедура отправляет такое сообщение.

Обеспечение безопасности приложения

К настоящему моменту мы полностью реализовали функциональные возможности приложения: произведено тестирование программы как со стороны администратора, так и со стороны конечного пользователя — все работает замечательно. Осталось обеспечить безопасность. Если не предусмотреть какой-либо либо процедуры регистрации, управляющей доступом к страницам администратора, то это приведет к тому, что каждый пользователь сможет проникнуть туда и получить информацию об адресах электронной почты подписчиков, послать фиктивные сообщения по электронной почте, уничтожить списки рассылки и т. д. При создании любого общедоступного web-приложения обеспечение безопасности должно быть одной из самых первых вещей, которые принимаются во внимание на стадии разработки. Однако для небольших приложений, подобных создаваемому нами, обеспечение безопасности и аутентификация пользователей может быть реализована в качестве завершающего этапа разработки — без каких-либо проблем и с потрясающей легкостью.

Установки для обеспечения безопасности в файле web.config

Впервые мы столкнулись с файлом `web.config`, когда обсуждали вопрос о хранении строки соединения. Этот файл конфигурации может быть использован также для хранения установок web-страниц, управления состоянием сеанса, приложения (как например, включенных нами строк `ConnString` и `SettingsFile`), трассировки, компиляции, обеспечения безопасности и для многого другого. Этот файл оказывается очень полезным, поскольку он позволяет переопределять установки IIS. Мы планируем рассмотреть хранящиеся в этом файле установки для обеспечения безопасности. ASP.NET. Названный файл установок предоставляет простой способ реализовать гибкую систему аутентификации пользователей (в результате чего пользователю будет либо разрешен, либо запрещен доступ к сайту) и систему определения полномочий (которая либо предоставляет какие-либо права, либо отбирает их у пользователей). Основная часть работы выполняется в следующих строках программы:

```
<configuration>
   <system.web>
      <authentication mode="Forms">
         <forms name=" .NEWSMAILER" loginUrl="Login.aspx"
            protection="All" timeout="60">
            <credentials passwordFormat="Clear" >
               <user name="Admin" password="AdminOK"/>
               <user name="Marco" password="MarcoOK"/>
            </credentials>
         </forms>
      </authentication>

      <authorization>
         <deny users="?" />
      </authorization>
   </system.web>

   <appSettings>
      <add key="connstring" value="Provider=Microsoft.Jet.OLEDB.4.0;
         Data source=E:\Projects\NewsMailer\DB\NewsMailer.mdb" />
   </appsettings>
</configuration>
```

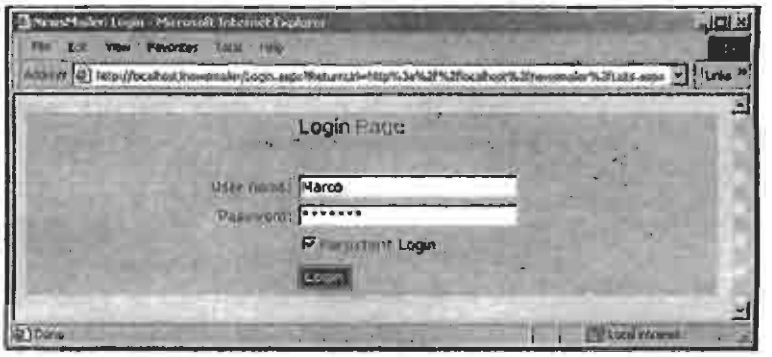
Выполнение аутентификации в режиме Forms означает, что для регистрации будет использоваться web-форма (в качестве альтернативных вариантов можно указать систему HTTP Basic или систему Microsoft Passport). Строки программы, расположенные внутри раздела `<authentication>`, определяют URL страницы регистрации, на которую пользователи будут отправляться, если они не пройдут процедуру аутентификации при попытке доступа к закрытым страницам. Для хранения информации о регистрации и о полномочиях пользователей, которым разрешен доступ к сайту, используется куки (`.NEWSMAILER`). Для нашего простого примера мы храним полномочия в виде комбинаций имени пользователя — пароля. В завершение тег `<deny users="?" />`, расположенный внутри раздела `<authorization>` определяет, что анонимным пользователям запрещается входить в закрытую часть приложения.

Обеспечение безопасности в ASP.NET может быть намного более сложным и профессиональным — это всего лишь пример основ системы аутентификации пользователей и определения полномочий. Если вы собираетесь пользоваться файлом `web.config`, то у вас также появляется возможность зашифровывать

пароли с помощью Triple DES. Другая возможность заключается в том, чтобы либо хранить имена пользователей и пароли в таблице базы данных и использовать собственную функцию для проверки правильности регистрационной информации, переданной пользователем, либо проверять пользователей по Windows Active Directory. Все это, однако, выходит за рамки настоящего раздела. Более подробную информацию вы сможете почерпнуть в документации .NET Framework SDK или в книге "ASP.NET для профессионалов" ("Professional ASP.NET"), выходящей в издательстве "Лори" в 2004 г.

Файл Login.aspx и его код поддержки

Теперь нам необходимо создать страницу `Login.aspx`, которая в законченном виде выглядит так, как показано на рисунке справа.



Код, необходимый для создания страницы, не представляет из себя ничего особенного — для других страниц нам уже приходилось писать более сложные программы, поэтому сразу перейдем непосредственно к коду, ответственному за обработку нажатия кнопки Login:

```
protected void LoginUser_Click(object sender, EventArgs e)
{
    // проверка имени и пароля данного пользователя
    if (FormsAuthentication.Authenticate(UserName.Text, Password
    {
        // если пользователь аутентифицирован, разрешить вход
        // (если если это требует пользователь) перейти на страницу,
        // которую требует процедура аутентификации
        FormsAuthentication. RedirectFromLoginPage(UserName.Text,
            Persistent.Checked);
    }
    else
    {
        // если имя пользователя или пароль неверны, вывести сообщение об ошибке
        InvalidLogin.Visible = true;
    }
}
```

В этой программе осуществляется вызов метода `Authenticate()`, который производит проверку правильности имени пользователя и пароля. Если они не распознаны, то выводится надпись InvalidLogin, в противном случае, регистрационная информация записывается в куки (если в окне выбора Persistent Login установлен соответствующий флажок), и браузер переходит на страницу, которая необходима для регистрации пользователя. URL этой страницы передается в строке запроса, и вы можете увидеть ее в линейке с адресом на рисунке, приведенном выше: там выводится URL `localhost/newsmailer/Lists.aspx` в закодированном виде. В этом случае, если пользователь попытается загрузить некоторую страницу и не пройдет процедуру аутентификации, то он будет снова отправлен на страницу регистрации и после успешного прохождения аутентификации возвращен на исходную страницу.

Теперь приложение выглядит вполне завершенным, однако остается еще одна маленькая проблема. Что произойдет, если пользователь отправит форму, для того

чтобы подписаться на некий список? Такой пользователь не будет аутентифицирован, поэтому он будет отправлен на страницу регистрации. Такого, естественно, быть не должно, и одним из возможных решений этой проблемы является перенос страницы `Subscribe.aspx` в другую папку, куда анонимный пользователь будет иметь возможность загрузить требуемую страницу. Такой способ будет, конечно, работать, но здесь потребуется предусмотреть два файла `web.config`: один со строкой соединения, хранящийся в корневой папке, а второй с установками для обеспечения безопасности, хранящийся в закрытой папке. Существует, однако, более простое решение, для реализации которого требуется добавить лишь несколько строк в уже существующий файл `web.config`, и это позволит оставить страницу `Subscribe.aspx` там же, где находятся все остальные. Мы располагаем возможностью задавать различные опции для обеспечения безопасности разных файлов с помощью тега `<location>`; это означает, что можно закрыть все файлы, как мы делали это раньше, а затем открыть анонимным пользователям доступ только к странице `Subscribe.aspx`:

```
<configuration>
   <system.web>

...Code omitted for clarity

   </system.web>

   <location path="Subscribe.aspx">
      <system.web>
         <authorization>
            <allow users="*" />
         </authorization>
      </system.web>
   </location>

   <appSettings>
      <add key="connstring" value="Provider=Microsoft.Jet.OLEDB.4.0;
         Data source=E:\Projects\NewsMailer\DB\NewsMailer.mdb" />
      </appSettings>
</configuration>
```

Теперь при отправке пользователем формы и загрузке страницы `Subscribe.aspx` ASP.NET распознает пользователей как анонимных и разрешает им доступ к этой странице. Если же пользователь осуществляет попытку попасть на любую другую страницу (возможно, узнав правильный URL этой страницы), то у него будет запрошена регистрационная информация.

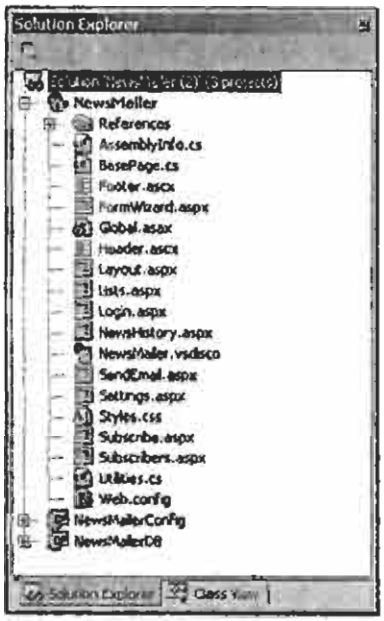
И еще один момент: что необходимо предпринять, если возникает необходимость выйти из системы и уничтожить куки, в которых хранится регистрационная информация? Мы воспользуемся методом `SignOut()` объекта `FormsAuthentication` (который находится в пространстве имен `System.Web.Security`) — он выполняет именно те действия, которые нам требуются. Названный метод вызывается в процедуре `Page_Load()`, если в строке запроса параметру `Action` присваивается значение `"logout"`:

```
protected void Page_Load(object sender, EventArgs e)
{
   // если строка QueryString имеет выходной параметр, осуществите выход
   // и удалите куки для входной страницы
   if (Request.Params["Action"]=="logout")
      FormsAuthentication.SignOut();
}
```

Страница с таким значением параметра вызывается посредством пункта меню Logout, который описан в пользовательском управляющем элементе `Header` следующим образом:

```
<asp:HyperLink runat="server"
  NavigateUrl="Login.aspx?ReturnUrl=Lists.aspx&Action=logout
  Text="Logout" Class="MenuItem" .../>
```

Обратите внимание на то, что в этом случае, кроме параметра `"Action=logout"` используется также и параметр `ReturnUrl`, имеющий значение `Lists.aspx`. Это оказывается необходимым, поскольку после выхода из системы мы возвращаемся обратно к регистрационной форме. Если администратор сразу повторно зарегистрируется, то странице регистрации необходимо иметь информацию о том, на какую страницу необходимо перейти.



Компиляция модуля

Последним шагом, перед тем как приступим к тестированию приложения и загрузке его на web-сервер, является компиляция всех файлов с кодом поддержки, созданных до настоящего момента. Если вы используете VS.NET, то ваш Solution Explorer должен выглядеть так, как показано на рисунке слева.

Выберите команду **Build**, и VS.NET откомпилирует ваш проект, позаботившись о ссылках на два бизнес-модуля (`NewsMailerConfig` и `NewsMailerDB`).

Запуск приложения

После того как модули откомпилированы, необходимо убедиться в том, что они находятся в директории `bin`, которая, в свою очередь, помещается в корневой папке web-приложения или виртуальной папке, создаваемой администратором IIS. Измените строку соединения в файле `web.config` так, чтобы она указывала на БД Access (или DSN), и после этого вы готовы к тестированию программы в Internet Explorer. Чтобы попасть на страницу `Lists`, необходимо отправиться по адресу `http://localhost/newsmailer/Lists.aspx`.

Проблемы совместимости браузеров

Все страницы указанного приложения были разработаны и протестированы в Internet Explorer 5.5. Если вы используете приложение совместно с Netscape 6, то его внешнее представление будет несколько отличаться, поскольку Netscape интерпретирует некоторые установки таблицы стилей по-другому. При использовании более ранних версий Netscape данный сайт будет выглядеть еще хуже. Это составляет большую проблему для сайтов, которые должны быть доступны для широкой публики. В промышленных системах, когда невозможно потребовать от пользователей применения определенного браузера (как, например, в данном случае — по крайней мере, в отношении администраторов), представляется важным разрабатывать web-страницы так, чтобы они могли работать на любом браузере.

Итоги

Первое приложение ASP.NET, использующее базу данных, завершено. Главной целью этого практического примера было показать, насколько простым является написание web-приложения, как для этого можно использовать C# и что нужно для того, чтобы воспользоваться преимуществами, которые дает .NET Framework. Мы узнали, что использование серверных управляющих элементов ASP.NET позволяет запросто сэкономить на написании нескольких сотен строк кода, получив при этом более стройный и приятный результат.

Настоящее приложение может использоваться прямо в таком виде для множества web-сайтов, однако можно включить в него дополнительные возможности, чтобы сделать его еще более мощным. Предлагаем вам несколько идей:

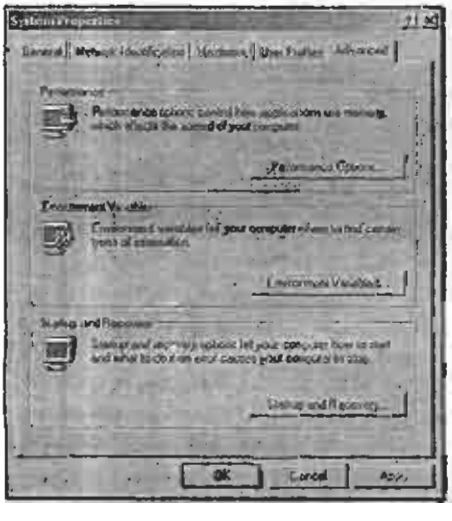
- Предусмотрите возможность создавать и работать с подписными формами, которые позволяют пользователю подписываться за один прием сразу на несколько списков рассылки, сообщив свой адрес электронной почты только один раз.
- Предусмотрите возможность, позволяющую пользователю выбирать, хочет ли он получать новостные сообщения в виде HTML или обычного текста, и отправлять различные сообщения в соответствии с этим выбором.
- Предусмотрите возможность отправлять вместе с новостными сообщениями различные приложения.
- Предусмотрите возможность работы нескольких администраторов, каждый из которых обладает различными правами. Например, администратор базового уровня получает возможность только просматривать текущие списки рассылки и информацию о подписчиках, администратор второго уровня может отправлять новые сообщения, а самый продвинутый администратор обладает полным набором прав, то есть возможностью добавлять, удалять и изменять списки рассылки и информацию о подписчиках.

Большинство тем, рассмотренных здесь, посвящены web-разработке с использованием ASP.NET, но многие принципы и некоторый код (в частности, класс `Utilities`) могут быть также приспособлены для использования в приложениях Windows, например, управление вводом/выводом, объектно-ориентированная разработка, доступ к БД посредством ADO.NET и многие классы .NET Framework. Поэтому вы можете воспользоваться этим кодом, даже если web-программирование не является основным объектом ваших интересов.



Задание переменной среды PATH

Для того чтобы иметь возможность использовать компилятор C# и другой инструментарий .NET из командной строки Windows, необходимо убедиться в том, что переменная среды `PATH` соответствующим образом изменена. В командной строке поиск файлов осуществляется в рамках текущей директории; если же возникает необходимость запустить файл из какой-либо другой директории, то для этого требуется задать информацию, где именно Windows следует искать требуемый файл. Это может быть сделано с помощью переменной среды `PATH`. В ней содержится список имен директорий, разделенных точкой с запятой, где Windows будет осуществлять поиск запрашиваемого файла, если его не удается обнаружить в текущей директории. Необходимо добавить в эту переменную информацию о двух папках, для того чтобы система Windows могла отыскать все инструменты .NET; одна из них — это `C:\Program Files\Microsoft.Net\FrameworkSDK\Bin`, в которой находится часть инструментов .NET, используемых из командной строки, такие например, как `ildasm`, `gacutil` и `sn`; а вторая — это папка `C:\WINNT\Microsoft.Net\Framework\v1.0.2914` (цифры в имени последней поддиректории могут варьироваться в зависимости от того, какая конкретно версия .NET Framework инсталлирована на вашей машине), именно в ней расположен собственно компилятор C# (`csc`), а также библиотека классов `System`.

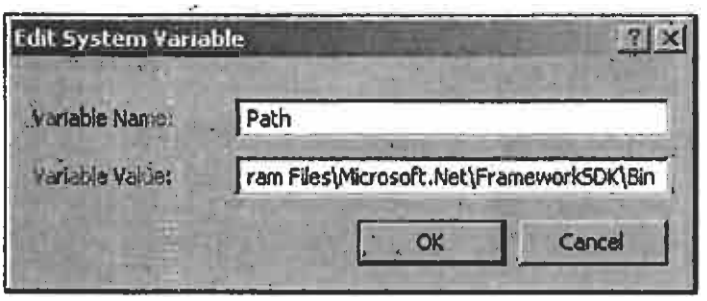


Windows 2000/XP

Чтобы задать переменную `PATH` на компьютере, работающем под Windows 2000 или XP, откройте окно **System** в **Control Panel** и щелкните на вкладке **Advanced** (см. рис. слева).



Перейдите на вкладку **Advanced**, щелкните мышью на кнопке **Environment Variables...** и прокрутите окно со списком **System variables** до переменной `Path` (см. рис. справа).



Выберите ее и щелкните мышью на кнопке Edit..., которая расположена под окном со списком.

Введите в окно Variable Value полный путь к директориям, которые вы хотите добавить в конец списка директорий, хранящегося в переменной среды PATH, как показано на рисунке слева. Переменная PATH представляет собой разделенный точками с запятой список, поэтому требуется включить одну точку с запятой перед названием первого пути, а вторую — между названиями двух путей. Например, для современной версии .NET Framework требуется добавить к концу строки следующий текст:

```
;C:\Program Files\Microsoft.Net\FrameworkSDK\Bin;C:\WINNT\
Microsoft.Net\Framework\v1.0.2914
```

Чтобы закрыть диалог Edit System Variable, щелкните мышью на кнопке OK, а затем щелкните еще два раза для закрытия диалогов Environment Variables и System Properties. После этого изменения будут сохранены в переменной PATH. Если вы ранее открыли окно командной строки, то придется закрыть и открыть его заново, поскольку новое значение недоступно для уже существующих окон.

Windows NT



В заданин значения переменных среды под Windows NT4 существует несколько незначительных отличий. Откройте диалоговое окно System из Control Panel точно так же, как в Windows 2000, а затем щелкните мышью на вкладке Environment (см. рис. справа).

Выберите переменную Path из окна со списком System Variables. Текущее значение этой переменной появится в окне со списком Value; добавьте необходимые для .NET Framework значения пути в конец записи точно так же, как и для Windows 2000 (также разделенные точкой с запятой). Щелкните мышью на кнопке Set для обновления имеющегося значения, а для его сохранения — на кнопке OK. Теперь новое значение будет доступно для всех вновь открываемых окон.

Windows 95/98/ME

Если вы используете Windows 9x или Windows ME, то, чтобы изменить значение переменной PATH, требуется модифицировать текстовый файл autoexec.bat. Этот файл содержит информацию, используемую для конфигурирования Windows в момент ее запуска, и располагается в корневой директории жесткого диска (например, C:\autoexec.bat). Откройте файл в редакторе Notepad или любом другом текстовом редакторе и добавьте следующую строку:

```
SET PATH=%PATH%;C:\Windows\Microsoft.NET\Framework\v1.0.2914
```

В данном случае нужно привести значение пути в соответствие с используемой системой и версией .NET Framework. Введенный путь будет добавлен к существующей переменной PATH. Обратите внимание на то, что на машинах, работающих под управлением Windows 9x или Windows ME, будут инсталлированы не все инструменты .NET Framework. В завершение необходимо перезапустить Windows — только после этого сделанные изменения вступят в силу.



Инсталляция MSDE

Предназначенные для работы при больших нагрузках системы управления БД (database managemant system — DBMS) типа Oracle и Microsoft SQL Server позволяют поддерживать выполнение десятков тысяч транзакций в минуту, при этом сокращая до минимума время отключения для проведения работ по сопровождению и внесению изменений. Они представляют собой замечательный пример современного программного обеспечения, однако за эту мощь приходится платить определенную цену. Прежде всего — дороговизной таких систем DBMS, которая начинается с несколько тысяч долларов и круто идет вверх при увеличении возможностей. Кроме того, за применение систем DBMS промышленного использования приходится также расплачиваться памятью, мощностью процессора и объемом памяти на жестком диске, которые необходимы для работы таких систем.

С точки зрения разработчиков установка такой полномасштабной системы на переносных компьютерах для тестирования SQL-операторов может оказаться невозможной. Компания Microsoft предлагает определенную альтернативу — она взяла продукт SQL Server и убрала все инструменты, предназначенные для управления и ведения разработок, оставив только ядро, отвечающее за работу с данными. Затем они ограничили максимальное число пользователей — не более пяти, и объем данных — 2 гигабайта. Полученный в результате продукт получил название MSDE (Microsoft Data Engine — ядро для работы с данными компании Microsoft). Он вполне может использоваться в качестве серверной части, к которой совершенно естественно могут добавляться клиентские с CD, что позволяет инсталлировать менее масштабные варианты SQL Server.

Инсталляция Microsoft Data Engine (MSDE)

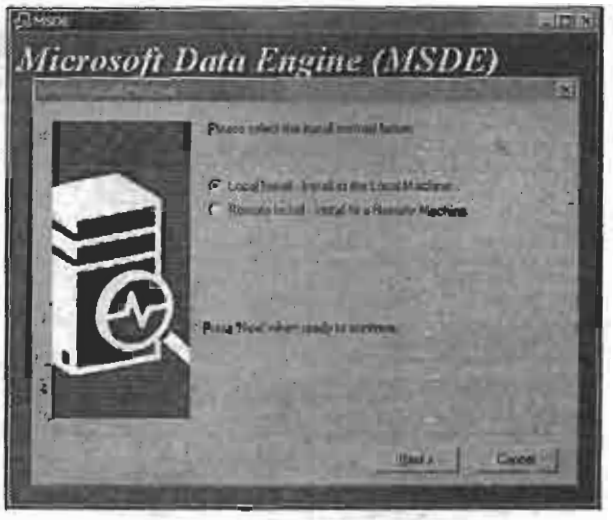
На дисках Office 2000 программа MSDE располагается на `Disk One\SQL\x86\SETUP\SetupSQL.exe`.

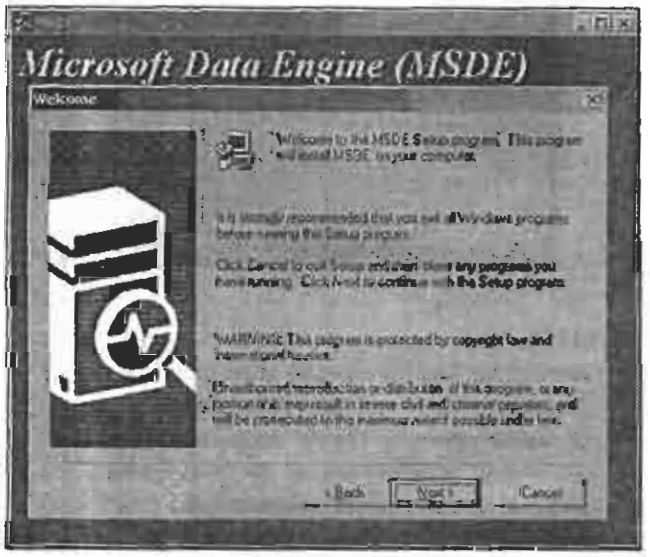
Если вы обладаете лицензией на какой-либо продукт Visual Products, то вы можете скачать MSDE с сайта `http://msdn.microsoft.com/vstudio/msde/default.asp` или — по тому же самому адресу — сделать заказ, и CD будет выслан вам по почте. Инсталляция продукта очень проста; требуется 50 Мбайт памяти на жестком диске плюс собственно ваши данные. И хотя это всего лишь MSDE, а не Microsoft SQL Server, приложение будет инсталлировано в папке `C:\MSSQL7`. Внутри этой папки располагается папка `Data`, в которой вы сможете обнаружить базы данных, устанавливаемые по умолчанию.

1. Весь процесс инсталляции занимает менее пятнадцати минут. Начните с завершения всех работающих приложений.

2. С помощью пункта меню **Start | Run | browse** найдите выполняемый файл `SetupSQL.exe` и запустите его. Для этого необходимо один раз щелкнуть мышью на кнопке **OK**, чтобы покинуть окно поиска, и второй раз щелкнуть на кнопке **OK** для запуска файла установки.

3. Будет запущен мастер инсталляции, который попросит выбрать метод инсталляции (см. рис. справа). Выберите опцию **Local Install** (локальная инсталляция) и щелкните мышью на кнопке **Next**.

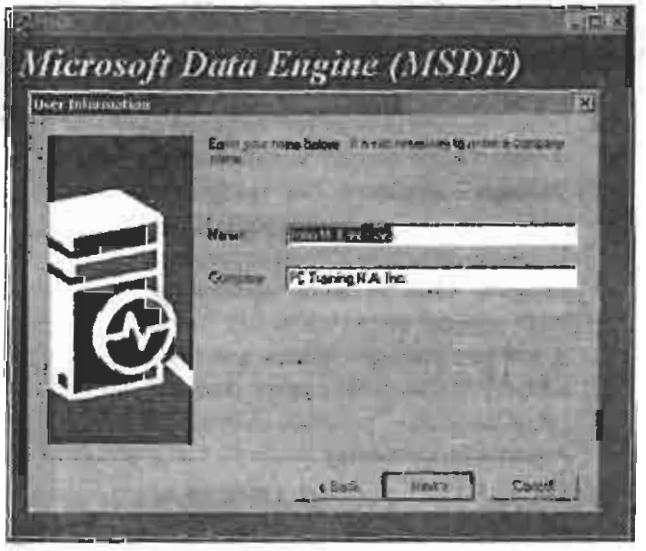


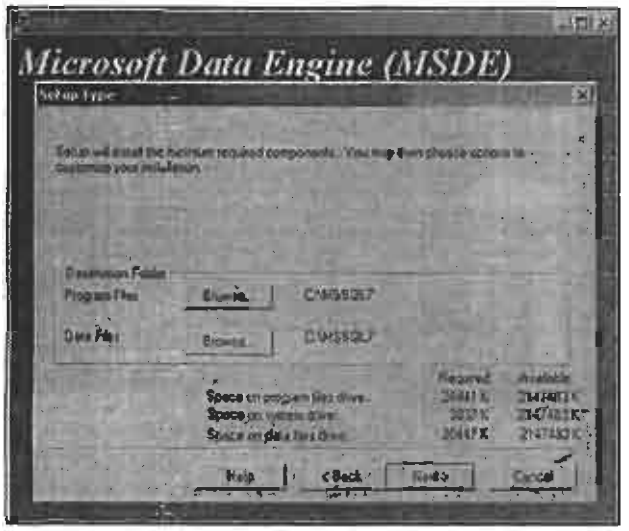


4. Теперь мастер выводит экран, представленный на рисунке слева. На этом экране содержится предостережение о необходимости завершить все другие приложения, а также информация, касающаяся охраны прав. Щелкните мышью на кнопке **Next**.

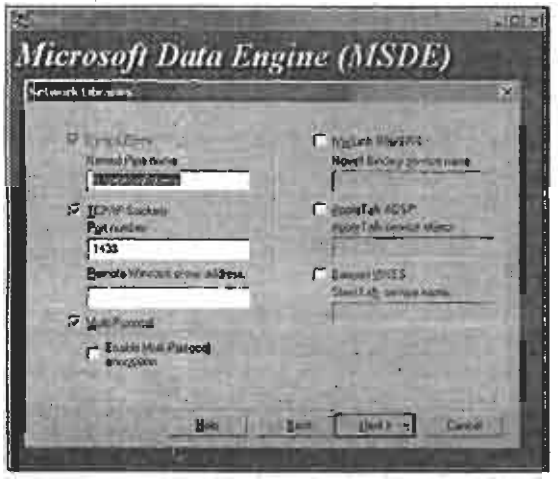
5. Теперь мастер запросит информацию, касающуюся вашего имени и вашей компании (см. рис. справа). Введите требуемую информацию и щелкните мышью на кнопке **Next**.

6. Теперь нужно определить **Setup Type** (тип установки) — это единственное место, где можно изменять местоположение файлов. Как и для полномасштабного продукта SQL Server, в данный момент можно разместить свои данные на диске,





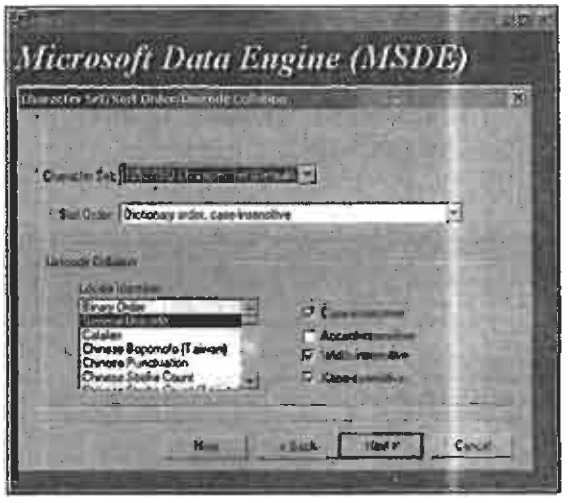
по используемому языку программирования (см. рис. справа). Все это не требует пояснений. Обычно достаточно принять значения, используемые по умолчанию. Щелкните мышью на кнопке **Next**.



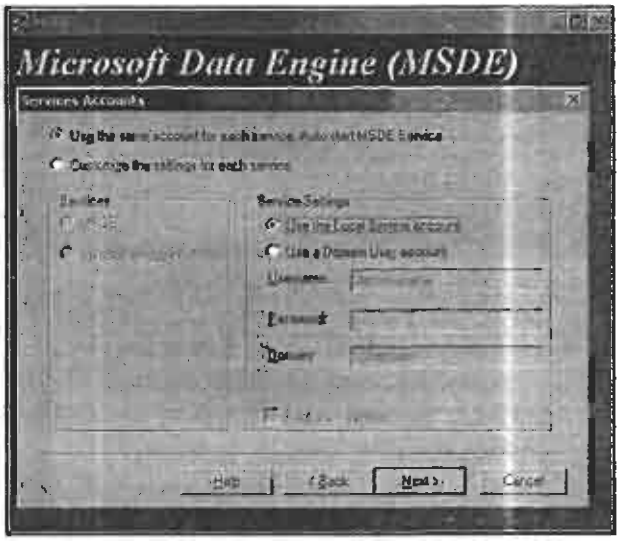
9. Далее мастер потребует информацию о том, какой способ управления учетными записями ему следует использовать (см. рис. справа). В самом простом случае следует выбрать **'Use the same account for each service'** (использование одной и той же учетной записи для всех служб) и **'use local system account'** (использование учетной записи локальной системы).

отличном от того, на котором располагаются программные файлы (см. рис. слева). Щелкните мышью на кнопке **Next**.

7. Для работы MSDE необходимо располагать информацией об используемом по умолчанию наборе символов, а также о некоторых других установках



8. MSDE предоставляет возможность инсталлировать несколько библиотек для установления сетевых соединений (см. рис. слева). В самом простом случае воспользуйтесь значением, предлагаемым по умолчанию, которое предполагает инсталляцию только **TCP/IP Sockets** и **Multi-Protocol**. Щелкните мышью на кнопке **Next**.





10. В этот момент мастер выведет экран для запуска процедуры копирования (см. рим. слева); щелкните мышью на кнопке Next.

11. После этого увидите окно с сообщением, которое информирует о том, какие нменно файлы копируются. Далее выводятся последовательные сообщения по мере развития процесса.

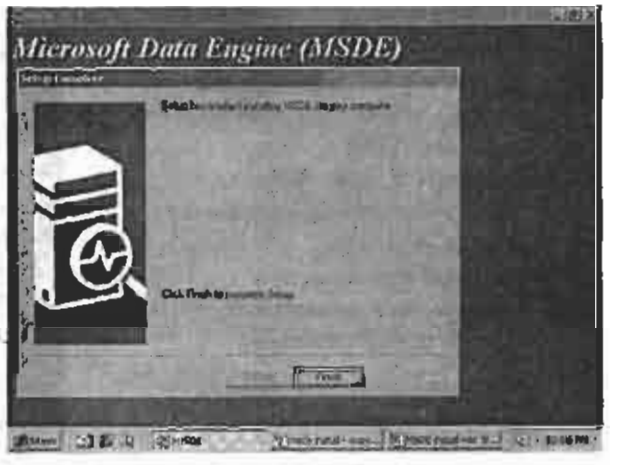
12. Мастер выводит экран 'setup complete' (начальная установка завершена) (см. след. ниже рис.). Щелкните мышью на кнопке Finish.

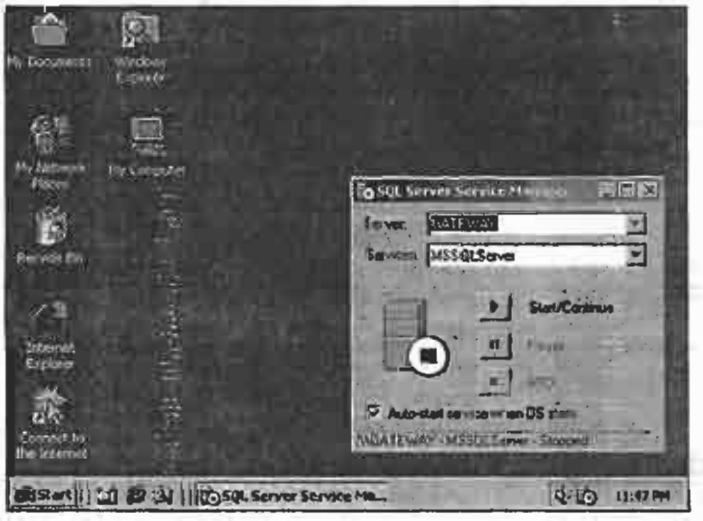
13. Теперь можно запустить MSDE, выбрав пункт меню
Start | programs | MSDE | Service Manager

Service Manager для MSDE ничем не отличается от Service Manager для полного продукта SQL Server.

14. При желании можете указать режим автозапуска.

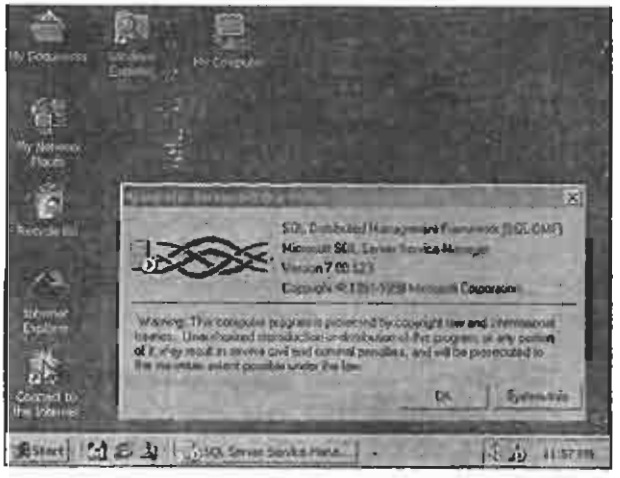
Для работы MSDE требуется приблизительно 5,3 Мбайт оперативной памяти, а для Service Manager — еще 2,3. Поэтому установка режима автозапуска приведет к захвату соответствующих ресурсов, что нежелательно, если вы на самом деле не собираетесь использовать систему. Вы всегда при необходимости можете запустить ее вручную.





15. Для запуска службы MSDE щелкните мышью на кнопке Start с зеленой стрелкой.

В процессе работы Service Manager с правой стороны системной линейки, расположенной внизу экрана, будет находиться вспомогательная иконка. Пока MSDE выполняется, она выводится с зеленой стрелкой, а после остановки — с красным квадратом, как показано на рисунке слева.

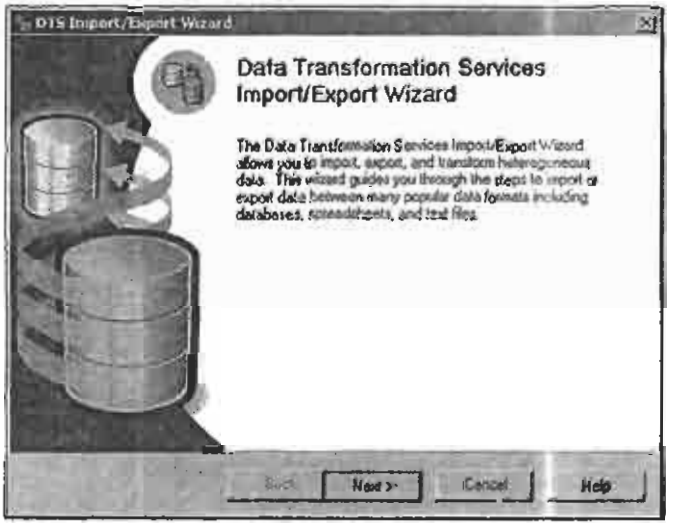


Чтобы проверить номер используемой версии, можное щелкнуть правой кнопкой мыши на вспомогательной иконке MSDE и выбрать пункт меню About (О программе). Вы должны получить сообщение о том, что используете SQL Service Manager версии 7 (см. рис. слева).

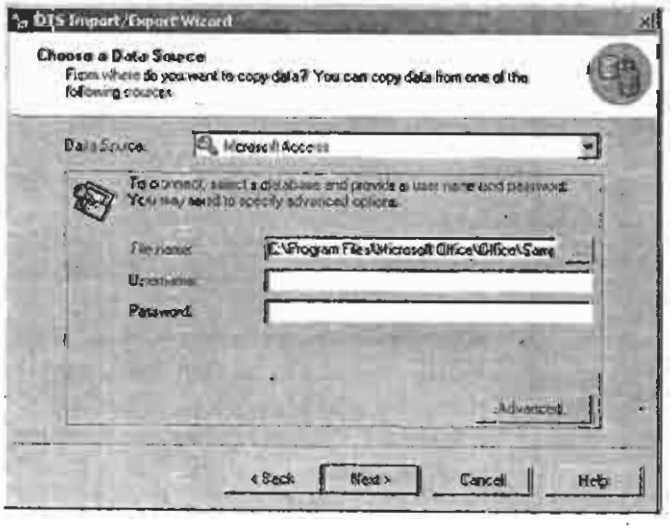
Импорт Northwind в MSDE

MSDE инсталлируется без каких-либо стандартных баз данных, поэтому требуется импортировать базу данных, чтобы работать с примерами, приведеннымн в главах 18 и 19. MSDE и SQL Server поставляются совместно с утилитой, предназначенной для импорта и экспорта данных. Мы воспользуемся предоставляемыми ею возможностями для импорта в MSDE повсеместно распространенной базы данных `Northwind` (`nwind.mdb`), которая входит в комплект поставки Access и Visual Basic 6.

1. Определите местоположение БД `Northwind`, воспользовавшись пунктом меню Start | Search | Files and folders | File и набрав текст `nwind.mdb` в окне Named:. Скопируйте (но ие переносите) этот файл в папку с именем `C:\BegCSharp\AppendixB` (которую вам придется создать самостоятельно).



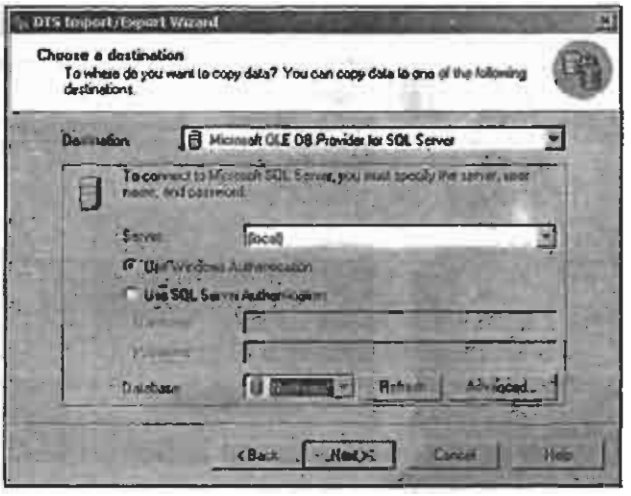
2. Перейдите в раздел меню Start/programs/MSDE or SQL Server/Import and Export Data. Перед вами возникнет экран, представленный на рисунке справа,

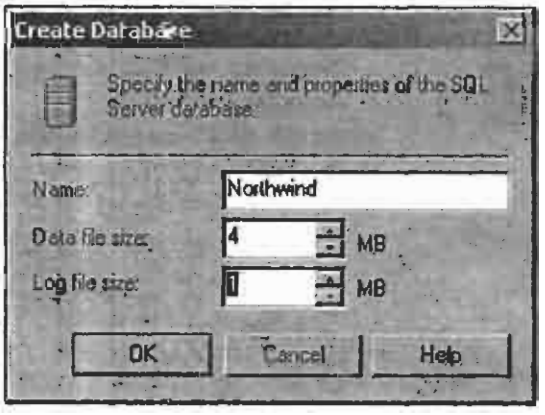


Data Transformation Wizard (мастер трансформации данных).

3. Щелкните мышью на кнопке Next, чтобы перейти к экрану Source (источнпк), и выберите в качестве значения поля Data Source (источник данных) Microsoft Access, как показано на рисунке слева, и задайте значение в поле File name (имя файла) — `nwind.mdb` (необходимо сослаться на копию `Northwind`, которую вы создали в директории `C:\BegCSharp\AppendixB`).

4. Щелкните мышью на кнопке Next, чтобы перейти к экрану Destination (назначение) (см. рис. справа) и выбрать для поля Destination значение Microsoft OLE DB Provider for SQL Server (провайдер OLE DB для SQL Server компании Microsoft), для поля Server — значение (local), а для Username (имя пользователя) в разделе SQL Server Authentication (аутентификация SQL Server) — значение `sa` (пароля не требуется).

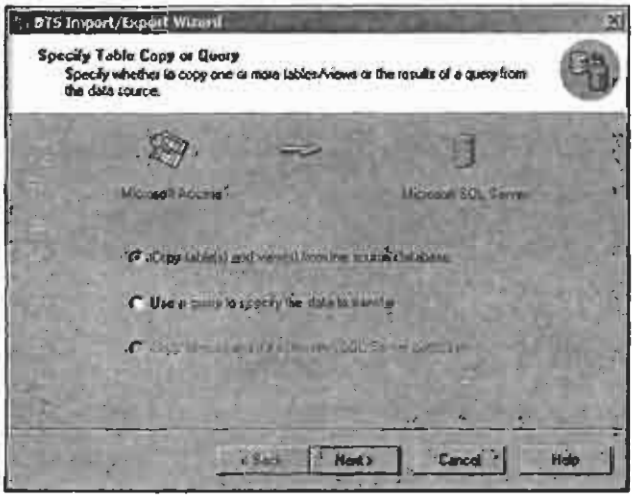


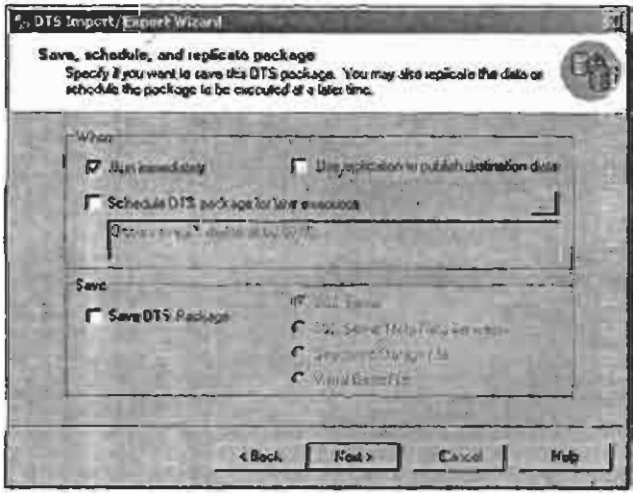


в этом диалоге следует ввести имя базы данных (мы будем использовать в качестве имени "`Northwind`") и задать размер файла Data, равным 4 Мбайт, а файла Log — 1 Мбайт.

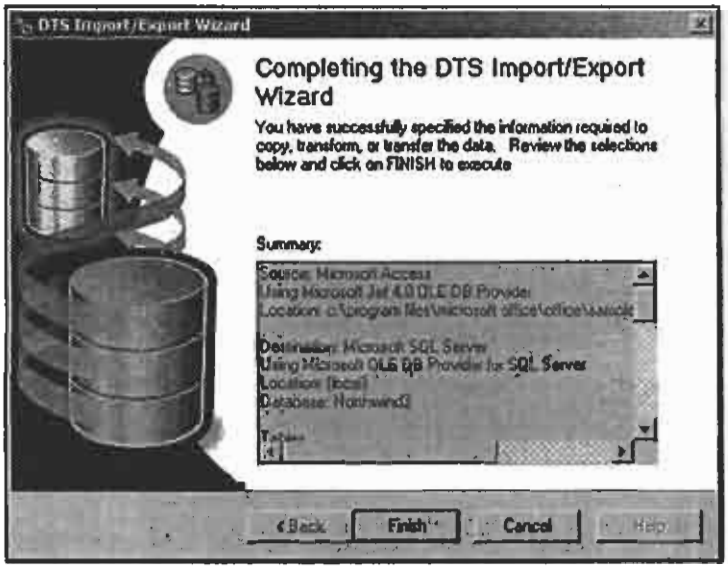
5. Для перехода на экран Method (метод) (см. рис. справа) щелкните мышью на кнопке Next и выберите copy table (копировать таблицу). Щелкните на кнопке Next, а затем на следующем раскрывшемся экране выберите copy all (копировать все).

Перед тем как щелкнуть мышью на кнопке Next, выберите для базы данных значение **<new>**, что позволит перейти в показанный на рисунке слева диалог Create Database (диалог создания базы данных);





6. Щелкните мышью на кнопке Next и выберите режим Run immediately (немедленный запуск), как показано на рисунке слева.

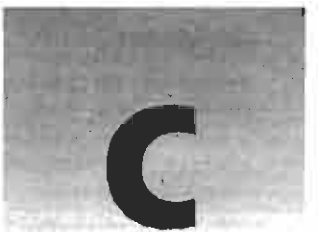


Щелкните мышью на кнопке Next еще раз, взгляните на окончательный результат (см. рис. слева) и щелкните мышью на кнопке Finish.

7. Дождитесь появления сообщения об успешном завершении, а затем щелкните на кнопке Done (сделано).

8. Обратите внимание на то, что все запросы, ответы и прочая информация копированию не подлежат.

9. Откройте Windows Explorer и убедитесь, что в директории `C:\MSSQL7\Data` теперь находится файл `Northwind.mdf` (а не `.mdb`).



Дополнительные возможности для компиляции C#

В настоящем приложении приводится список различных возможностей компилятора C#, которыми можно воспользоваться при выполнении компиляции проектов C# без использования возможностей Visual Studio.NET, либо, если нужно, чтобы компилятор выполнил какие-либо операции, не поддерживаемые Visual Studio.NET. Эти опции представлены в виде нескольких таблиц, где они сгруппированы в соответствии со своей категорией.

В первой таблице приводятся различные форматы файлов, которые могут включаться в выходной поток компилятора:

Опция	Назначение
`/doc:<имяфайла>`	Выполняется обработка комментариев, предназначенных для создания XML-документации (помеченных тремя слэшами — ///), и выходной поток направляется в указанный файл.
`/nooutput`	Осуществляется компиляция кода, но не создается выходной файл; этот режим может оказаться полезным для отладки, поскольку в нем можно увидеть все предупреждения и сообщения об ошибках на консоли.
`/out:<имяфайла>`	Позволяет задавать имя выходного файла. Если файл не задан, то компилятор создает стандартный `.exe`-файл с тем же именем, что и исходный файл (за исключением расширения).
`/target:<опция>` `/t:<опция>`	Позволяет задавать формат выходного файла. У параметра `<опция>` существует четыре возможных значения: • `exe`: создается стандартный выполняемый файл (это значение используется по умолчанию). • `library`: создается программная библиотека (DLL). • `module`: создается программный отсек (модуль, у которого отсутствует манифест), который в дальнейшем добавляется к модулю (с использованием `/addmodule`). • `winexe`: создается выполняемый файл Windows.

Во всех случаях, кроме использования опции `/target:module`, в создаваемый EXE-файл будет включаться манифест (или в первую DLL, если EXE-файл не создается). Заметим, что `/target` может быть сокращено до `/t`.

В следующей таблице приводятся опции, которые могут задаваться в командной строке для определения режима оптимизации при компиляции.

Опция	Назначение
/optimize<+ | -> /o<+ | ->	Устанавливает или отменяет режим оптимизации, выполняемой компилятором, чтобы создать меньший по объему, более быстрый и более эффективный выходной код. По умолчанию такой режим отменен. Для включения режима следует воспользоваться следующим синтаксисом: /optimize или /optimize+ Для отмены режима используется синтаксис: /optimize-

В следующей таблице представлены опции для создания модулей .NET и обращения к ним:

Опция	Назначение
/addmodule:<имя отсека>	Позволяет задавать один или нескольких программных отсеков, которые будут включены в указанный модуль. Если задается более одного отсека, они отделяются друг от друга точкой с запятой. Эта опция не может использоваться в Visual Studio.NET.
/nostdlib<+ | ->	Позволяет задать, следует или нет импортировать стандартную библиотеку (mscorlib.dll), которая импортируется по умолчанию. Если вы желаете реализовать собственное пространство имен System и классы, то можно не импортировать стандартную библиотеку. Для этого используется следующий синтаксис: /nostdlib или /nostdlib+ Синтаксис для импорта стандартной библиотеки: /nostdlib-
/reference:<имя модуля> /r:<имя модуля>	Предназначен для импорта метаданных из файла модуля. Можно либо задать полный путь к данному модулю, либо этот путь может быть задан в переменной среды PATH, либо указать путь относительно текущего проекта. Если задается более одного файла, то они разделяются точками с запятой.

В следующей таблице объясняются опции, которые применяются при отладке и поиске ошибок:

Опция	Назначение
/bugreport:<имя файла>	Создает файл с заданным именем, в котором содержится вся информация об ошибках, сгенерированная компилятором. В состав содержимого файла входит: • Копия исходного файла. • Список опций компилятора. • Информация о версии компилятора, об операционной системе и т. д. • Весь выходной поток компилятора • Описание обнаруженной проблемы и возможное решение (при наличии такового) Эта опция не может использоваться в Visual Studio.NET.

Продолжение таблицы

Опция	Назначение
`/checked<+ | ->`	Позволяет определять, будет ли переполнение целых значений приводить к возникновению ошибки во время выполнения. Это относится только к тому коду, который находится вне области действия описателей checked и unchecked. По умолчанию этот режим отключен. Синтаксис, включающий режим контроля за переполнением, имеет вид: `/checked` **или** `/checked+` Для отказа от контроля за переполнением следует использовать следующий синтаксис: `/checked-`
`/debug<+ | ->` `/debug:<опция>`	Позволяет генерировать отладочную информацию. Для этой цели используется синтаксис: `/debug` **или** `/debug+` Для отмены режима используется следующий синтаксис: `/debug-` По умолчанию режим отладки не используется. Если задается вывод отладочной информации, то имеются два варианта, касающиеся того, какого типа отладочная информация будет генерироваться: • `/debug:full`: позволяет присоединить отладчик к работающей программе • `/debug:pdbonly`: позволяет осуществлять отладку исходного кода, если программа запускается в отладчике, но будет выводить только ассемблер, если выполняющаяся программа присоединена к отладчику.
`/fullpaths`	Задает полный путь к файлу, в котором содержится ошибка. Эта опция недоступна в Visual Studio.NET.
`/nowarn:<номер>`	Не позволяет компилятору генерировать сообщения об ошибках с заданным номером. Номер предупреждения, отказ от вывода которого задается, определяется опцией <номер>. Если задается более одного номера, то они разделяются запятыми. Эта опция недоступна в Visual Studio.NET.
`/warn:<опция>` `/w:<опция>`	Позволяет задавать минимальный уровень выводимых ошибок. Имеются следующие возможности: • 0: Отказ от вывода всех ошибок. • 1: Вывод сообщений только о грубых ошибках. • 2: Вывод сообщений о грубых ошибках и об ошибках средней "тяжести". • 3: Вывод сообщений о грубых ошибках, об ошибках средней "тяжести" и о незначительных ошибках. • 4: Вывод сообщений обо всех ошибках, включая информационные сообщения
`/warnaserror<+ | ->`	Устанавливает режим, в котором все предупреждения трактуются как ошибки. Для включения режима используется следующий синтаксис: `/warnaserror` **или** `/warnaserror+` Для отмены режима следует использовать синтаксис: `/warnaserror-` По умолчанию режим не устанавливается.

Опция	Назначение
/define<имя> /d<имя>	Определяет символ препроцессора с именем <имя>.

Таблица слева демонстрирует, как следует задавать директивы препроцессора.

В следующей ниже таблице представлены опции, связанные с использованием внешних ресурсов:

Опция	Назначение
/linkresource:<имя файла> /linkres:<имя файла>	Позволяет создавать ссылку на указанный .NET-ресурс. Имеет следующие два необязательных дополнительных параметра (разделенных запятыми): • identifier: Логическое имя ресурса; имя, которое будет использовано для загрузки ресурса (по умолчанию используется <имя файла>). • mimetype: Строка, представляющая тип носителя, на котором находится ресурс (по умолчанию отсутствует). Эта опция не может использоваться в Visual Studio.NET.
/resource:<имя файла> /res:<имя файла>	Внедряет указанный .NET-ресурс в выходной файл. Имеет следующие два необязательных дополнительных параметра (разделенных запятыми): • identifier: Логическое имя ресурса; имя, которое будет использовано для загрузки ресурса (по умолчанию используется <имя файла>). • mimetype: Строка, представляющая тип носителя, на котором находится ресурс (по умолчанию отсутствует).
/win32icon:<имя файла>	Позволяет включить заданный файл с иконкой Win32 (.ico) в выходной файл.
/win32res:<имя файла>	Позволяет включить заданный файл с ресурсом Win32 (.res) в выходной файл. Эта опция недоступна в Visual Studio.NET.

В заключительной таблице представлены остальные опции компилятора:

Опция	Назначение
@<имя файла>	Позволяет задавать файл, где содержатся все опции компилятора и исходные файлы, которые будут обрабатываться компилятором таким образом, как если бы они были заданы в командной строке.
/baseaddress:<адрес>	Позволяет задавать наиболее предпочтительный базовый адрес, по которому будет загружаться DLL. Значение опции <адрес> может быть представлено в десятичном, шестнадцатеричном или восьмеричном виде.
/codepage:<идентификатор>	Позволяет задавать страницу кодировки (значение, передаваемое опцией <идентификатор>), которая должна использоваться для всех файлов с исходным кодом в процессе компиляции. Эта опция оказывается полезной, когда для файлов с кодами на C# используется набор символов, отличный от используемого в системе по умолчанию. Эта опция недоступна в Visual Studio.NET.
/help /?	Помещает перечень опций компилятора в стандартный выходной поток. Эта опция недоступна в Visual Studio.NET.

Продолжение таблицы

Опция	Назначение
`/incremental<+ | ->` `/incr<+ | ->`	Позволяет осуществлять последовательную компиляцию файлов с исходным кодом, при которой компилируются только функции, в которые были внесены какие-либо изменения с момента предшествующей компиляции. Информация о состоянии предшествующей компиляции сохраняется в двух файлах — в файле с расширением `.dbg` (или в файле с расширением `.pdb`, если установлен режим `/debug`), предназначением для хранения отладочной информации, и в файле с расширением `.incr` — здесь сохраняется информация о состоянии. Для задания такого режима можно воспользоваться одним из двух следующих вариантов синтаксиса: `/incremental` или `/incremental+` Чтобы отменить этот режим, следует использовать следующий синтаксис: `/incremental-` По умолчанию этот режим не установлен.
`/main:<класс>`	Позволяет задавать местоположение основного метода `Main()`, если в исходном коде таких методов несколько.
`/nologo`	Позволяет отказаться от вывода "шапки" с информацией о компиляторе. Эта опция недоступна в Visual Studio.NET.
`/recurse:<директория\файл>`	Позволяет осуществлять поиск файлов, подлежащих компиляции, внутри поддиректорий. Имеет две опции: • `<директория>` (необязательная): определяет директорию или поддиректорию, с которой следует начинать поиск. Если эта опция не задана, то поиск осуществляется с директории текущего проекта. • `<файл>`: определяет файл или файлы, поиск которых следует осуществлять.
`/unsafe`	Позволяет выполнять компиляцию кода, который помечен ключевым словом `unsafe`.

Beginning C#
Karli Watson, Marco Bellinaso, Ollie Cornes, David Espinosa
Zach Greenvoss, Christian Nagel, Jacob Hammer Pedersen
Jon Reid, Matthew Reynolds, Morgan Skinner, Eric White

Copyright © 2001
All rights reserved

C#
Карли Ватсон, Марко Беллиназо, Олли Корнс, Дэвид Эспиноза
Зах Гринфосс, Кристиан Нейджел, Джейкоб Хаммер Педерсен
Джон Рейд, Мэттью Рейнольдс, Морган Скиннер, Эрик Уайт

Переводчик И. Штерев
Научный редактор Н. Смольянинов
Корректор Л. Белая
Верстка Т. Кирпичевой

Все права защищены. Никакая часть данной книги не может воспроизводиться, размещаться в поисковых системах или пересылаться в любой форме и любыми средствами без письменного согласия издателя, за исключением использования коротких цитат в критических статьях и обзорах.

Автор и издатель при подготовке этой книги сделали все от них зависящее для обеспечения правильности содержащейся в ней информации. Однако информация, помещенная в данной книге, продается без каких-либо гарантий — прямых или косвенных. Ни авторы, ни издательство Wrox Press, ни его дилеры и дистрибьюторы не могут быть признаны ответственными за любые прямые или косвенные убытки, которые были вызваны или могли быть вызваны данной книгой.

Copyright © 2001 Wrox Press Ltd.
Arden House, 1102 Warwick Road, Acocks Green,
Birmingham, B27 6BH, UK

ISBN 1-861004-98-2

© Издательство "Лори", 2005

Изд. № : OAI (03)
ЛР № : 070612 30.09.97 г.
ISBN 5-85582-228-1

Подписано в печать 20.12.2004 Формат 84 ×108/16
Гарнитура Литературная Печать офсетная
Печ. л. 55 Тираж 1500 Заказ № 17/04
Цена договорная

Издательство "Лори"
123100, Москва, Шмитовский пр., д. 13/6, стр. 1 (пом. ТАРПЦАО)
Телефон для оптовых покупателей: (095)256-02-83
Размещение рекламы: (95)259-01-62

WWW.LORY--PRESS.RU