Вячеслав Понамарев

Программирование на C++ /C# в Visual Studio .NET 2003

- **Платформы .NET и .NET Framework**
  Основные понятия
- **Среда Visual Studio .NET 2003**
  Рекомендации по настройке и расширению среды разработки
- **Языки C++ и C#**
  Примеры создания программ
- **Web-приложения**
  Создание полноценного Web-магазина компьютерной техники

МАСТЕР_ПРОГРАММ_

Вячеслав Понамарев

Программирование на C++ /C# в Visual Studio .NET 2003

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068+800.92C++/C#
ББК 32.973.26-018.1
П56

Понамарев В. А.

П56 Программирование на C++/C# в Visual Studio .NET 2003. — СПб.: БХВ-Петербург, 2004. — 352 с.: ил.

ISBN 5-94157-402-9

В книге рассматриваются особенности разработки приложений в среде Visual Studio .NET 2003 с применением языков программирования C++ и C#. Проводится сравнительный анализ алгоритмических языков. Показано удобство создания приложений на языке C#, а также аналогичность этого языка языку Visual Basic .NET. Раскрываются основы работы с данными, изучаются технологии ADO и XML. Для повышения наглядности материала используются схемы, рисунки, таблицы, приводятся также поясняющие программы-примеры. Уделяется внимание самостоятельному созданию больших проектов с последующим ознакомлением с механизмом отладки приложений.

Для программистов

УДК 681.3.068+800.92C++/C#
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Яковлева*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.01.04.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 28,38.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-402-9

Содержание

Введение

Новая технология .NET постепенно внедряется в большинство языков программирования, которые разрабатываются не только корпорацией Microsoft. Но основу составляют именно языки, входящие в состав Microsoft Visual Studio .NET. В книге будет рассматриваться программирование на двух основных языках: C# и C++ из новой среды разработки Microsoft Visual Studio .NET 2003. В большинстве случаев упор будет делаться на язык C#, т. к. он является на сегодняшний день наиболее соответствующим технологии .NET.

В процессе работы использовалась среда разработки Microsoft Development Environment 2003 Version 7.1.3088. Все примеры, приведенные в книге, тестировались именно в этой версии.

На кого рассчитана книга

Книга предназначена для всех желающих, стремящихся изучить особенности языка программирования C#, который активно продвигается корпорацией Microsoft. Книга рассчитана на подготовленных программистов, знакомых с языками программирования C или C++.

Я благодарен всем читателям, которые присылают свои рецензии о моих книгах. Присылайте свои отзывы и пожелания по этой и другим книгам, написанным мною, по адресу в Интернете: **ponamarev@mail.ru**.

Структура и особенности книги

Книга состоит из шести частей.

Часть I кратко описывает платформы .NET и .NET Framework. Раскрывается общая модель программирования в среде .NET. Изучается механизм работы с компонентами в среде .NET.

В *части II* описывается механизм настройки среды разработки. Раскрываются принципы расширения среды разработки. Изучается работа с кодом. Описывается работа с файлами ресурсов, а также проектами и решениями.

Часть III рассказывает об языках C++ и C#. Приводится краткое описание синтаксиса двух языков. Рассматриваются примеры небольших программ на языках C++ и C#. Проводится сравнительный анализ языков и описываются основные возможности данных языков программирования.

Часть IV посвящена раскрытию основ работы с наборами данных, технологиями ADO и XML.

Часть V посвящена созданию больших проектов в среде Visual Studio .NET 2003. Рассматриваются принципы создания Web-приложений на языках C++ и C#. На одном большом примере описывается создание Web-магазина. Изучаются механизмы отладки приложений.

В *части VI* (приложениях) вы найдете дополнительную информацию.

Приложение 1 содержит ссылки на сайты, которые посвящены технологии .NET и языку C#.

Приложение 2 объясняет различие между редакциями Visual Studio .NET 2003.

Приложение 3 содержит аппаратные и программные требования, которые предъявляются при установке Visual Studio .NET 2003.

Приложение 4 приводит список языков, которые позволяют компилировать программы в IL-код.

Приложение 5 содержит краткое описание типов данных, используемых в языках .NET.

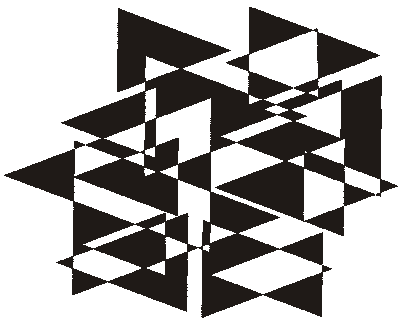
Наконец, в *приложении 6* приводится краткий список сокращений и терминов, которые используются в книгах и справочных материалах, посвященных технологии .NET.

Соглашения, используемые в книге

В книге были использованы следующие издательские соглашения:

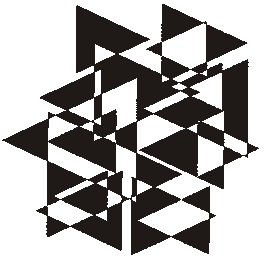
- листинги, идентификаторы, имена функций, классов, переменных, команд, которые вы можете увидеть на экране монитора, выделены моноширинным шрифтом;
- *курсивом* выделены новые термины, которые впервые встречаются в тексте книги. А также таким образом помечены важные места текста;
- **полужирным** шрифтом выделены имена элементов интерфейса: окон, пунктов меню, вкладок и т. д.;
- важные моменты, на которые стоит обратить внимание, выделены в примечания;
- названия клавиш клавиатуры заключены в треугольные скобки, например <F1>. Для иллюстрации одновременного нажатия нескольких клавиш используется символ "+", например <Ctrl>+<Alt>+<O>.

ЧАСТЬ I



Платформа .NET

В этой части раскрываются основы платформ .NET и .NET Framework. Описывается общая модель программирования, языковая интеграция, а также язык CLR. Кроме того, в этой части вы найдете общее описание работы с компонентами .NET.



ГЛАВА 1

Платформы .NET и .NET Framework

В этой главе приводятся сведения о платформах .NET и .NET Framework. Описываются цели создания и преимущества платформ .NET и .NET Framework.

Что такое Microsoft .NET

Первое заявление корпорации Microsoft о переходе на платформу .NET было летом 2000-го года на конференции PDC 2000 в городе Орландо штата Флорида.

Платформа .NET создавалась как среда разработки приложений под Windows с новым интерфейсом программирования. В .NET интегрированы службы компонентов COM+, работа с XML, среда Web-разработки ASP, а также ориентация на создание интернет-приложений и поддержка новейших Web-сервисов (SOAP, UDDI и WSDL).

Вся платформа .NET состоит из различных продуктов, которые можно условно разделить на четыре группы:

- *средства разработки* — т. е. языки программирования (Visual C, C#, Visual Basic .NET, Visual Java), среда Common Language Runtime (CLR, общеязыковая среда выполнения), библиотека классов для создания разнообразных приложений, а также инструментальная среда разработки Visual Studio .NET;
- *Web-сервисы* — т. е. возможность применения коммерческих Web-сервисов (таких, как .NET MyServices), которые необходимы для создания Web-приложений, требующих идентификации пользователей;
- *специализированные серверы* — набор серверов SQL Server, Exchange Server, BizTalk и др., объединенных в одно семейство серверов .NET Enterprise Servers. Эти серверы обеспечивают работу с базами данных, с электронной почтой, и многое другое;

- *поддержка устройств* — встроенная поддержка новых устройств ("умных" устройств), которые могут работать с технологиями .NET (например, мобильные телефоны).

Архитектура платформы .NET

В упрощенном общем виде платформа .NET состоит из пяти основных компонентов (рис. 1.1).

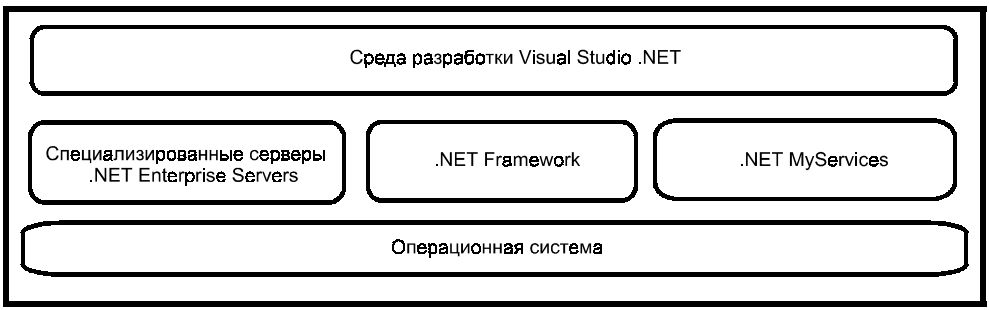


Рис. 1.1. Архитектура платформы .NET

На нижнем уровне платформы находится операционная система (Windows XP, Windows 2000, Windows ME или Windows CE). Кроме перечисленных операционных систем может использоваться и другое специализированное программное обеспечение для так называемых "умных" устройств.

На уровне, расположенном выше уровня операционных систем (рис. 1.1), находятся сразу три компонента:

- *специализированные серверы .NET Enterprise Servers* — набор серверных продуктов, таких как Application Center, BizTalk Server, Commerce Server, Exchange Server, Host Integration Server, Internet Security Acceleration Server и SQL Server;
- *набор Web-сервисов .NET MyServices* — представляющих собой готовые блоки кода, которые разработчик может включать (за дополнительную плату) в свои проекты;
- *.NET Framework* — новая инфраструктура разработки и исполнения Windows-приложений, которая включает в себя общеязыковую среду выполнения CLR, а также общую структуру классов, которые можно использовать в любом языке программирования, относящемся к семейству .NET.

На верхнем уровне архитектуры .NET располагается среда разработки приложений Visual Studio .NET. *Visual Studio .NET* — это интегрированная среда

разработки (Integrated Development Environment, IDE), которая содержит разнообразные средства для создания приложений на языках .NET.

Цели создания .NET Framework

Одной из главных целей создания .NET Framework является упрощение построения Windows-приложений.

Для реализации таких приложений было создано большое количество сред разработки, каждая из которых отличалась от других библиотеками классов, синтаксисом, программным интерфейсом приложений (API, Application Programming Interface) и др. Никакой стандартизации между этими интерфейсами и библиотеками классов просто не существовало.

Использование .NET Framework позволяет программисту использовать набор базовых классов, который подходит для любого языка программирования, относящегося к .NET (C++, C#, Visual Basic .NET и др.). Для успешной работы с любым из этих языков программирования не нужно всякий раз изучать новый API.

Перечислим основные цели создания .NET Framework:

- обеспечение одинаковых возможностей объектно-ориентированного программирования, независимо от используемого языка программирования;
- обеспечение неконфликтности версий создаваемого программного обеспечения и упрощение развертывания приложений, за счет регистраций общих DLL (Dynamic Link Library, динамически подключаемая библиотека) в глобальном кэше сборок (Global Assembly Cache, GAC);
- обеспечение безопасности выполнения кода приложения. .NET Framework предоставляет множество функций безопасности, обеспечивающих надежную защиту от несанкционированного проникновения в приложение или систему. В частности, вы можете защитить даже определенную часть исполняемого кода (например, метод).

Рассмотрим архитектуру .NET Framework.

.NET Framework

В общем виде, .NET Framework является простым системным приложением, работающим в операционной системе, такой как Windows (хотя, могут использоваться и другие операционные системы, например, относящиеся к семейству UNIX).

.NET Framework состоит из двух основных частей:

- *CLR* (Common Language Runtime, общеязыковая среда выполнения);
- *библиотеки классов .NET Framework.*

На рис. 1.2 представлена упрощенная схема архитектуры .NET Framework.

Главным компонентом .NET Framework является CLR. О среде CLR речь пойдет в *гл. 2.* Но пока отметим, что CLR занимается активизацией объектов, отладкой, обработкой исключений, проверкой типов, распределением памяти и ресурсов для объектов, а также осуществляет "сборку мусора", т. е. при прекращении использования того или иного объекта происходит автоматическое уничтожение объекта. Таким образом, CLR осуществляет те же самые действия, которые выполняет виртуальная Java-машина (Java Virtual Machine, JVM) в языке Java.

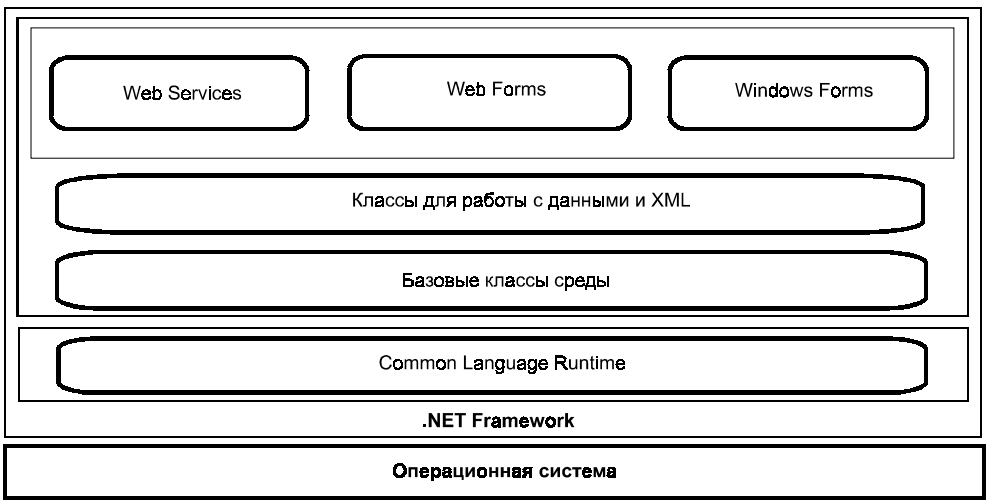


Рис. 1.2. Архитектура .NET Framework

CLR и JVM во многом похожи, но отличаются уже тем, что JVM поддерживает только язык программирования Java, а CLR поддерживает любые языки программирования, которые можно представить на общем для языков .NET промежуточном языке Microsoft Intermediate Language (MSIL, промежуточный язык Microsoft).

Другое отличие JVM и CLR заключается в том, что код Java с помощью JVM может работать на различных платформах (на которых установлена сама JVM), а код .NET работает пока только на платформе Windows.

Библиотеку классов .NET Framework можно условно поделить на несколько частей, согласно их предназначению.

Первая часть — набор базовых классов среды. Это стандартные классы, обеспечивающие работу со строками, ввод и вывод данных, многопоточность, работу с сетью, безопасность и многое другое. Базовые классы похожи по своему составу и функциональности на такие классы, как MFC, ATL и др.

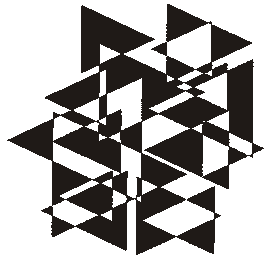
Вторая часть — это набор классов для работы с данными и XML (eXtensible Markup Language). Классы для работы с данными включают в себя класс работы с SQL (Structured Query Language, язык структурированных запросов), а также набор классов ADO.NET для манипулирования постоянными данными. Классы для работы с XML позволяют манипулировать XML-данными, а также выполнять XML-преобразования и XML-поиск.

Третья часть библиотеки расширяет возможности классов для работы с данными и XML и представляет три различных набора классов для поддержки трех технологий:

- *классы Web Services* — поддерживают разработку облегченных распределенных компонентов;
- *классы Web Forms* — позволяют быстро и эффективно создавать Web-приложения, в которых используется графический интерфейс пользователя (Graphical User Interface, GUI). Методы создания интерфейсов пользователя практически ничем не отличаются от создания интерфейсов пользователя на том же Visual Basic. Вы просто помещаете нужные элементы управления на Web-форму, задаете значения нужных свойств и создаете обработчики событий;
- *классы Windows Forms* — позволяют создавать обычные Windows-приложения, в которых используются стандартные элементы управления Windows. Вы можете рассматривать этот набор классов как улучшенную версию классов MFC.

Итак, мы кратко рассмотрели, из каких компонентов состоит .NET и .NET Framework. В *гл. 2* речь пойдет о такой важной части .NET Framework, как среда Common Language Runtime. Мы уточним, как в этой среде поддерживаются и исполняются .NET-компоненты, которые носят название *сборок* (assemblies).

ГЛАВА 2



Среда CLR

В этой главе речь пойдет о среде Common Language Runtime. Мы рассмотрим, что такое CLR и исполняемые файлы CLR. Кроме того, раскрывается понятие метаданных, а также сборок и манифестов.

Что такое Common Language Runtime

Общеязыковая среда выполнения (CLR), как уже упоминалось в *гл. 1*, входит в состав .NET Framework. CLR лежит в основе технологии .NET. Она создана, что называется, "с нуля". В основе CLR не лежит ни одна из ранее используемых в различных языках программирования библиотек, таких как MFC, ATL и др.

Среда CLR управляет выполнением кода, написанного на любом из языков семейства .NET. Данная среда занимается созданием различных объектов, выделяет под них системные ресурсы, выполняет проверку безопасности и исполняет код объектов. Кроме того, CLR отвечает за так называемую "сборку мусора", т. е. уничтожение неиспользуемых объектов и освобождению от них системных ресурсов.

Среда CLR оперирует так называемыми *сборками*.

Сборки (assemblies) — представляют собой .NET-компоненты, которые являются переносимыми исполняемыми файлами (Portable Executable, PE). Эти файлы являются файлами с расширениями exe или dll и состоят из метаданных и кода. О метаданных речь пойдет далее в этой главе, а пока рассмотрим исполняемые файлы CLR.

Исполняемые файлы CLR

Рассмотрим на небольшом примере (консольной программе, выводящей текстовое сообщение) особенности языков программирования C++, C# и Visual Basic .NET.

Пример на C++

Язык C++ в среде .NET несколько отличается от раннего C++ тем, что он расширился за счет нескольких специфичных ключевых слов. Это позволяет программам, написанным на C++ .NET, использовать новые возможности (такие, как "сборка мусора").

В листинге 2.1 приведен код простого консольного приложения, выводящего на экран строку текста. Заметьте, что данный код создается средой Visual Studio .NET автоматически, при создании нового консольного проекта при помощи диалогового окна **New Project** (рис. 2.1).

Листинг 2.1. Простое консольное приложение на C++

```
  //  это главный файл проекта для приложения VC++,
  //  сгенерированный мастером приложений (Application Wizard)
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System;
int _tmain()
{
   // TODO: замените код примера, расположенный ниже, на свой собственный
  Console::WriteLine(S"Hello World");
  return 0;
}
```

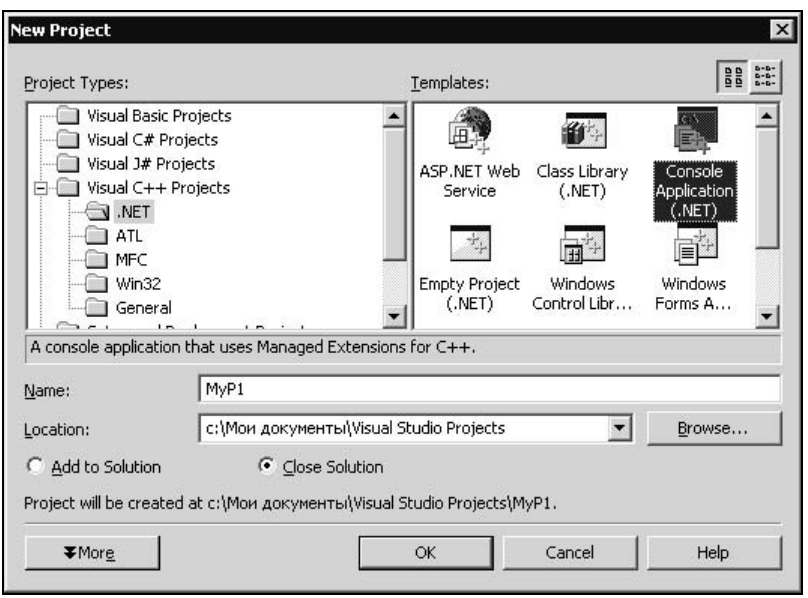


Рис. 2.1. Создание нового консольного приложения на C++

Попробуем разобраться с этим кодом. Как вы можете видеть, это вполне обычная программа на C++. Отличие заключается лишь в использовании директивы компилятора `#using` во второй строке кода. Данная директива делает доступными все типы из указанной библиотеки DLL. Директива `#using` может показаться аналогом директивы `#include`, но в отличие от нее импортирует типы любой сборки .NET, независимо от языка, на котором была написана эта сборка (естественно, это должен быть язык, относящийся к семейству .NET).

Далее все достаточно просто. В методе `_tmain` происходит вызов статического метода `WriteLine()` класса `Console`. Класс `Console` — это тип, который находится в библиотеке mscorlib.dll.

Кроме того, в следующей строке кода, с помощью оператора `using namespace`, компилятору сообщается, что будут использованы типы из пространства имен `System`. В противном случае код строки

```
Console::WriteLine(S"Hello World");
```

пришлось бы заменить на такое выражение:

```
System::Console::WriteLine(S"Hello World");
```

Примечание

Краткие описания пространств имен вы можете прочитать в *гл. 3*.

Если запустить это приложение (клавиша <F5> или пункт меню **Debug | Start**), то вы увидите результат (рис. 2.2).

Рис. 2.2. Результат выполнения консольного приложения

Пример на Visual Basic .NET

В листинге 2.2 приведен пример консольного приложения, выводящего одну строку текста на экран, написанного на языке Visual Basic .NET.

Листинг 2.2. Простое консольное приложение на Visual Basic .NET

```
Module Module1
Sub Main()
Console.WriteLine("Hello World")
End Sub
End Module
```

Как вы можете видеть, вызывается тот же самый метод `WriteLine()` класса `Console` (см. листинг 2.1). Если внимательно посмотреть на информацию о данной сборке — файл AssemblyInfo.vb, то будет виден следующий текст (листинг 2.3).

Листинг 2.3. Содержимое файла Assembly.vb

```
Imports System
Imports System.Reflection
Imports System.Runtime.InteropServices

' общая информация о сборке устанавливается с помощью следующего набора
' атрибутов. Вы можете изменить значения этих атрибутов для задания
' информации о сборке

' просмотр значений атрибутов сборки

<Assembly: AssemblyTitle("")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany("")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>

' следующий GUID является идентификатором ID библиотеки типов, если
' данный проект запланирован как COM-проект
<Assembly: Guid("B8980C3E-3F83-4174-A9CA-C4114ACD51F3")>

' информация о версии данной сборки состоит из следующих четырех значений
'
'   Major Version
'   Minor Version
```

```
'   Build Number
'   Revision
'
' вы можете указать все значения самостоятельно, или использовать
' значения по умолчанию, с помощью знака '*', как показано ниже

<Assembly: AssemblyVersion("1.0.*")>
```

Как видно из листинга 2.3, в самом начале программы происходит импорт типов из пространств имен `System`, `System.Reflection` и `System.Runtime.InteropServices`. Таким образом, используется то же самое пространство имен `System`, как и в примере на C++ (см. листинг 2.1).

Пример на C#

Рассмотрим реализацию консольной программы вывода текстового сообщения на языке C#, который был разработан фирмой Microsoft специально для среды .NET. Язык C# заимствовал синтаксис у языков C++ и Java. Это очень простой для понимания язык. Кстати, большинство утилит и классов .NET написаны именно на этом языке. Итак, пример простого консольного приложения на C# приведен в листинге 2.4.

Листинг 2.4. Простое консольное приложение на C#

```
using System;

namespace ConsoleApplication3
{
   ///  <summary>
   ///  общее описание класса Class1
   ///  </summary>
   class Class1
   {
     ///  <summary>
     ///  главная точка входа в приложение
     ///  </summary>
     [STAThread]
     static void Main(string[] args)
     {
        //
        //  TODO: стартовый код приложения добавьте здесь
        //
        Console.WriteLine("Hello World");
     }
   }
}
```

Здесь также используется пространство имен `System`. Отличие программы на C# от C++ заключается фактически лишь в том, что метод `Main()` в программе на C# является открытой статической функцией класса (в нашем примере — класса `Class1`), а не глобальной функцией, как в C++.

Если внимательно посмотреть на тексты листингов 2.2, 2.3 и 2.4, то можно заметить аналогию между языками Visual Basic .NET и C#. В Visual Basic .NET используются ключевые слова `Import` и `Module`, а в C# — `using` и `class`. Ну и конечно, в C# метод обозначается ключевым словом `void`, а в Visual Basic — `Sub`.

Переносимые исполняемые файлы .NET

Любой исполняемый файл операционной системы Windows (с расширениями exe или dll) должен соответствовать специальному формату *PE* (Portable Executable, переносимый исполняемый файл).

Формат PE является производным от формата *COFF* (Common Object File Format, общий объектный формат файлов).

Операционная система Windows "понимает" формат PE-файлов, поэтому может загружать и исполнять файлы с расширениями exe и dll. Таким образом, любой компилятор генерирует исполняемые файлы Windows в соответствии с форматами PE или COFF.

Обычные PE-файлы Windows можно разделить на две секции:

- *заголовки PE/COFF* (включая ссылки на содержимое PE-файла);
- *двоичные образы*. Включают в себя несколько секций:
  - .Data;
  - .Rdata;
  - .Rsrc;
  - .Text;
  - другие секции, созданные с помощью компилятора языка C++, используя директиву компилятора `pragma`. В этих секциях можно хранить, например, зашифрованные данные.

Рассмотрим внутреннее строение обычного PE-файла Windows (рис. 2.3).

Для поддержки новых возможностей среды CLR Microsoft немного изменила формат PE-файла, генерируемого средой .NET. А именно, были добавлены несколько новых секций (рис. 2.4).

Добавлены следующие секции:

- *заголовок CLR* — содержит информацию, указывающую, что PE-файл является исполняемым файлом .NET;

Рис. 2.3. Формат обычного PE-файла Windows

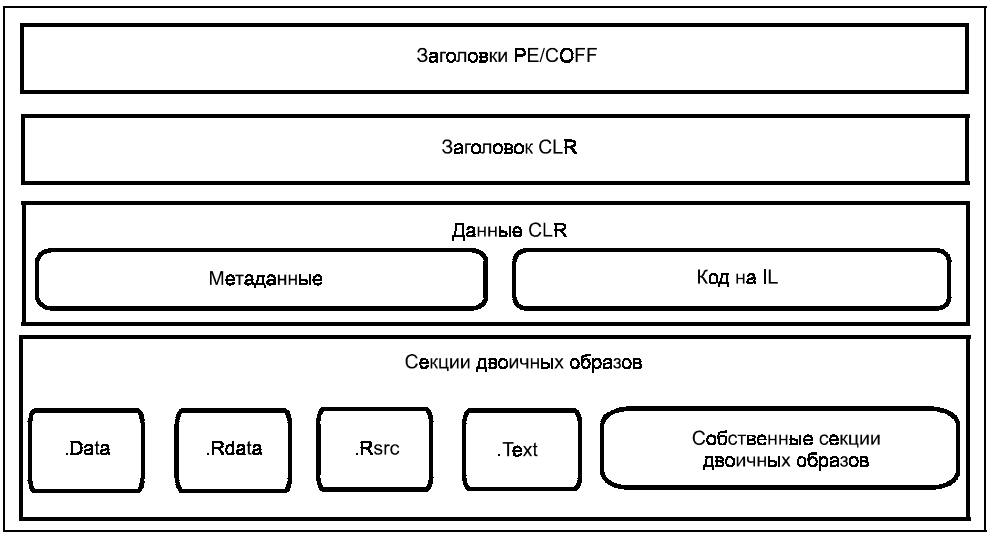


Рис. 2.4. Формат PE-файла, генерируемый средой .NET

- *данные CLR* — определяют, как будет выполняться программа. Состоят из:
  - *метаданных*;
  - *IL-кода*.

Рассмотрим теперь непосредственно метаданные и IL-код.

Метаданные

Метаданные (metadata) — это данные о данных. Другими словами, метаданные — это данные о ресурсах (например, к метаданным относятся определения типов, сведения о версиях, ссылки на внешние сборки и многое другое).

В среде .NET метаданные представляют собой механизм, применяемый всеми компиляторами .NET-языков и утилитами .NET. Таким образом, метаданные служат для описания всех типов, которые используются данной сборкой .NET. Метаданные детально описывают всю сборку (идентификатор сборки, ссылочные и экспортируемые типы, а также требования безопасности для исполнения сборки).

Ранее похожую информацию могли предоставлять библиотеки типов (type library) в технологии COM (Component Object Model, объектная модель компонентов). Но метаданные предоставляют больше информации. Они включают в себя еще многое другое (описание сборки и модулей, описание классов, методов, свойств, полей, событий, интерфейсов и др.).

Просмотр метаданных

Для просмотра метаданных и IL-кода PE-файла, созданного с помощью компилятора .NET, вы можете использовать специальную утилиту — IL-дизассемблер (ildasm.exe), который поставляется вместе с Visual Studio .NET 2003. Данная утилита по умолчанию располагается в папке, в которую вы установили Visual Studio .NET 2003, по следующему пути: **| SDK | v1.1 | Bin**. Например:

C: | Program Files | Microsoft Visual Studio .NET 2003 | SDK | v1.1 | Bin.

Примечание

Если у вас установлена более ранняя версия Visual Studio .NET, то путь к утилите ildasm.exe может несколько отличаться, например, так: **C: | Program Files | Microsoft Visual Studio .NET | FrameworkSDK | Bin**.

Запустим данную утилиту и откроем с ее помощью ранее созданного нами файла Hello.exe (рис. 2.5).

Как вы можете видеть, утилита ildasm.exe отображает метаданные PE-файла в иерархическом виде.

Щелкнем дважды левой кнопкой мыши на узле **MANIFEST**. Откроется дополнительное окно, в котором вы сможете увидеть такой текст (здесь он сокращен для экономии места):

```
.assembly extern mscorlib
{
...
}
.assembly extern Microsoft.VisualC
{
...
}
```

```
.assembly Hello
{
...
}
.module Hello.exe
  // MVID: {D69FDF78-C2FF-4884-B6C9-641AA3DC687F}
...
```

Первые две инструкции языка IL `.assembly extern` означают, что данный PE-файл ссылается на две внешние сборки: `mscorlib` и `Microsoft.VisualC`. Следующая инструкция `.assembly` описывает данную сборку `Hello`. Информация из блоков `.assembly` носит название манифестов.

Наконец, сразу после манифестов вы можете видеть инструкцию IL `.module`, которая сообщает имя модуля (Hello.exe).

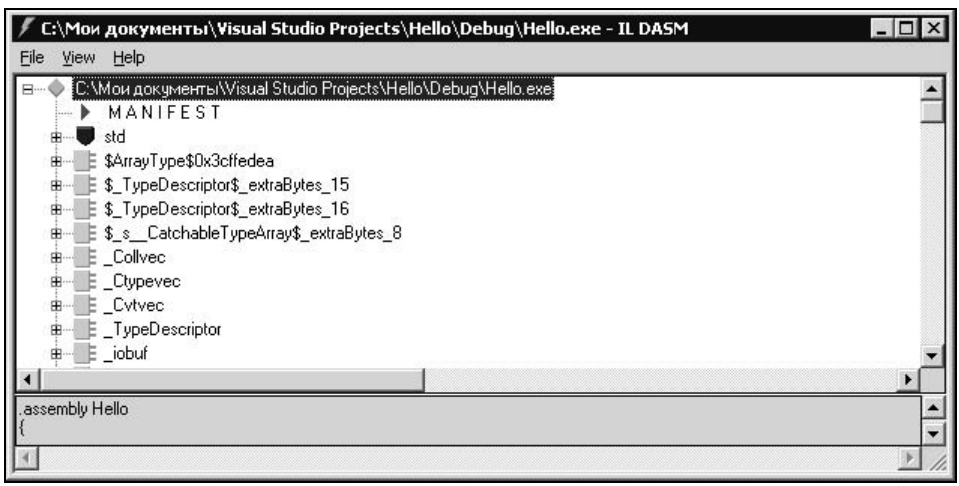


Рис. 2.5. Содержимое файла Hello.exe в утилите ildasm.exe

Рассмотрим еще один узел в иерархии нашего PE-файла, а именно:

main: int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)()

Щелкнув дважды левой кнопкой мыши на этом узле, вы можете видеть в открывшемся окне следующий текст:

```
.method public static int32
modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
main() cil managed
{
   .vtentry 1 : 1
```

```
    //  code size    14 (0xe)
   .maxstack 1
   IL_0000: ldstr "Hello World"
   IL_0005: call    void [mscorlib]System.Console::WriteLine(string)
   IL_000a: ldc.i4.0
   IL_000b: br.s    IL_000d
   IL_000d: ret
}   //  end of method 'Global Functions'::main
```

Как вы уже догадались, это непосредственно IL-код нашего метода `main()`, выводящего на экран строку текста.

Если вы хотите получить полную информацию о PE-файле в одном текстовом файле, то можете использовать комбинацию клавиш <Ctrl>+<D> в окне дизассемблера ildasm.

Таким образом, любой PE-файл, откомпилированный с помощью языков .NET, можно успешно просмотреть посредством данной утилиты.

Сборки и манифесты

Как уже упоминалось ранее, *сборки* представляют собой файл с расширениями exe или dll, а манифест — это описание или метаданные сборки (версия, имя, используемые другие сборки и т. д.).

Прежде термин "компонент" использовался для обозначения как классов COM, так и для COM-модулей (представляющих собой файлы с расширениями exe и dll). С появлением технологии .NET понятие сборки заменило собой понятие COM-модуля.

Для уникальности типов ранее (в технологии COM) использовались *глобальные уникальные идентификаторы* (GUID). Но таких идентификаторов было огромное количество, это идентификаторы приложений, библиотек, классов, интерфейсов и т. д. Все это затрудняло разработку приложений, т. к. в коде постоянно приходилось использовать числовые значения этих идентификаторов.

В технологии .NET ссылки на любой тип основываются на его имени и пространстве имен. Кроме того, для обеспечения уникальности используются единственные пары открытого и закрытого ключа (как в криптографии с открытым ключом).

Так как манифест сборки содержит информацию о ссылках на другие сборки, теперь отпадает необходимость хранения в реестре указаний по активизации компонентов и маршаллингу (как при использовании технологии COM). Загрузка необходимых типов из внешних сборок будет производиться, основываясь на манифесте сборки, тогда, когда эти типы понадобятся

приложению. Таким образом, можно уменьшить размер загружаемого приложения за счет применения небольших внешних сборок, на которые будут указаны ссылки в манифесте. Необходимые типы будут подгружаться из внешних сборок по мере надобности автоматически без перезагрузки системы или регистрации этих типов.

Сборки вообще являются одним из самых важных элементов технологии .NET. Сборки инкапсулируют в себе все типы, которые были в ней объявлены. Например, имеются две различные сборки: `Library` и `Shop`, в которых определен один и тот же тип `Book`. Для того чтобы обратиться к этому типу, необходимо указать сборку, из которой вы хотите получить данный тип. В противном случае среда CLR не будет знать, к какой сборке относится данный тип, и приложение просто не выполнится.

Рассмотрим теперь (как я и обещал ранее) манифест PE-файла Hello.exe, который мы уже немного изучали в этой главе, более подробно:

```
.assembly extern mscorlib
{
   .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
   .ver 1:0:5000:0
}
.assembly extern Microsoft.VisualC
{
   .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A)
   .ver 7:0:5000:0
}
.assembly Hello
{
   ...
   .hash algorithm 0x00008004
   .ver 1:0:1325:27733
}
.module Hello.exe
  //  MVID: {D69FDF78-C2FF-4884-B6C9-641AA3DC687F}
```

Как уже было отмечено ранее, наша сборка ссылается на две внешние сборки: `mscorlib` и `Microsoft.VisualC`. Первая сборка необходима для всех приложений, и она будет присутствовать в манифестах любых сборок, которые вы будете создавать.

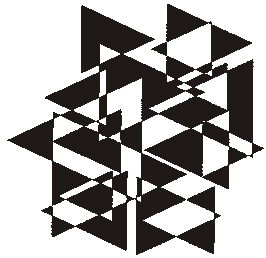
Обратите внимание на тот факт, что в каждой ссылке на эти две сборки имеется значение под названием `.publickeytoken`. Это значение генерируется компилятором, вычисляя хэш-функцию открытого ключа, связанного с каждой из этих сборок. Кроме того, после ключевого слова `.ver` имеется информация о версиях данных сборок.

Далее идет манифест для нашей сборки `Hello`. После ключевых слов `.hash algorithm` указан хэш-алгоритм, который применяется для генерации закрытого ключа. Кроме того, указан номер версии сборки.

Наконец, после ключевого слова `.module` указано имя файла для данной сборки. Заметим, что для данного модуля был сгенерирован GUID, который изменится, если вы перекомпилируете сборку. Таким образом, сравнивая значения GUID, можно утверждать, идентичны или нет два модуля с одинаковым именем.

На этом закончим рассмотрение среды CLR. Хотя это достаточно обширная тема для изучения, но она выходит за рамки нашей книги. В *гл. 3* будет рассказано об общей модели программирования в среде .NET.

ГЛАВА 3



Программирование в среде .NET

В этой главе мы рассмотрим общую модель программирования для среды .NET, а также небольшие примеры программ на базовых языках C++ и C#. Кратко рассмотрим пространства имен .NET Framework.

Общая модель программирования

Среда .NET предоставила программистам общие классы для всех языков программирования .NET. Ранее такое было невозможно. Для создания простого Windows-приложения разработчику предстояло сделать выбор между различными библиотеками (MFC, ATL, Win32 API и др.). Более того, знание классов, функций и интерфейсов, а также прочих возможностей одной библиотеки не могло помочь при использовании другой. Например, знание MFC не позволяло создавать программы на языке Visual Basic, использующем свою библиотеку VB.

В среде .NET достаточно знать, как реализуется та или иная функция на одном языке программирования, чтобы перенести этот код на требуемый язык программирования. Это стало возможным благодаря тому, что все классы, методы и все остальное в среде .NET имеют согласованное представление во всех языках программирования .NET.

Мы уже это видели, когда в *гл. 2* выводили на экран строку текста с помощью одного и того же метода `WriteLine()`, принадлежащего объекту `Console`.

Преимущества такого подхода очевидны. Вам практически не нужно изучать ничего нового при переходе с одного языка программирования платформы .NET на другой, естественно, за исключением синтаксиса самого языка программирования.

Все классы, которые вы можете использовать в любом языке программирования .NET, находятся в различных пространствах имен .NET Framework.

Каждое пространство имен содержит специфичные классы, объединенные по своему функциональному назначению. Основные пространства имен среды .NET Framework приведены в табл. 3.1.

Таблица 3.1. *Основные пространства имен среды .NET Framework*

Пространство имен	Краткое описание
`System`	Содержит все базовые классы, которые используются практически в каждой программе. Кроме того, включает и более сложные классы, такие как `AppDomain` и `GC`
`System.Collections`	Включает интерфейсы и классы, которые позволяют управлять различными коллекциями объектов, таких как списки, массивы, хэш-таблицы и словари
`System.IO`	Содержит набор классов, обеспечивающих синхронный и асинхронный ввод (вывод) потоков данных и работу с файлами
`System.Text`	Включает классы для работы с текстовыми данными, использующими различные кодировки (ASCII, Unicode, UTF-7 и UTF-8). Здесь же присутствуют абстрактные базовые классы для прямого и обратного преобразования блоков символов в блоки байтов
`System.Threading`	Предоставляет набор классов и интерфейсов, обеспечивающих создание многопоточных приложений
`System.Data`	Содержит классы для доступа к данным с использованием технологии ADO.NET
`System.Data.Common`	Включает в себя классы, обеспечивающие работу с провайдерами баз данных
`System.XML`	Предоставляет набор классов для работы с файлами формата XML
`System.Data.OleDb`	Содержит классы для соединения с наборами данных `OleDb`, выполнения команд над наборами данных и получения результата
`System.Data.SqlClient`	Включает набор классов для реализации программы-клиента SQL. Данные классы обеспечивают установление связи с SQL-сервером, выполнения команд и получения результата. Это пространство имен похоже на `System.Data.OleDb`, но было оптимизировано для работы с Microsoft SQL Server 7.0 или его более новой версией

Таблица 3.1 *(продолжение)*

Пространство имен	Краткое описание
`System.Data.SqlTypes`	Позволяет использовать "родные" типы данных SQL Server в приложениях. Использование этих типов данных помогает избежать ошибок преобразования типов и обеспечивает более высокое быстродействие приложений
`System.Web`	Включает набор классов и интерфейсов для создания Web-приложений, использующих HTTP (HyperText Transfer Protocol). Кроме того, содержит классы для передачи данных, работы с Cookies и многое другое
`System.Web.UI`	Содержит набор стандартных классов для создания графических интерфейсов пользователей Web-страниц
`System.Web.UI.HtmlControls`	Включает классы для специфичных элементов управления HTML, которые могут быть добавлены на Web-формы
`System.Web.UI.WebControls`	Содержит классы элементов управления для создания ASP.NET Web-серверов
`System.Web.Services`	Включает классы для использования Web-сервисов XML
`System.Windows.Forms`	Предоставляет классы для создания Windows-приложений, которые используют стандартные элементы управления операционной системы Windows
`System.Drawing`	Содержит классы, обеспечивающие возможность доступа к графической библиотеке GDI. Наибольшее число классов и возможностей предоставляют следующие подпространства имен, входящие в состав данного пространства: `System.Drawing.Drawing2D`, `System.Drawing.Imaging`, `System.Drawing.Text`
`System.ServiceProcess`	Содержит классы, позволяющие создавать приложения-сервисы (т. е. приложения, не имеющие интерфейса пользователя)
`System.ComponentModel`	Включает набор классов и интерфейсов, для работы с преобразователями типов, привязки к наборам данных, создания своих собственных компонентов и их лицензирования

Таблица 3.1 *(продолжение)*

Пространство имен	Краткое описание
`System.CodeDOM`	Содержит классы, которые могут использоваться для представления элементов и структуры исходного кода
`System.Diagnostics`	Включает классы, обеспечивающие возможность отладки приложений и трассировки выполнения кода. Кроме того, данное пространство имен содержит классы, позволяющие запускать системные процессы, читать и записывать системные события, а также отслеживать состояние системы
`System.DirectoryServices`	Предоставляет классы, которые позволяют получить доступ к Active Directory из кода приложения
`System.Management`	Включает классы для управления информацией и событиями, касающимися системы, устройств и приложений
`System.Messaging`	Содержит набор классов, позволяющих получать и отправлять сообщения по локальной сети
`System.Timers`	Позволяет использовать в приложении такие компоненты, как таймер (генерирующий событие всякий раз, по истечении некоторого временного интервала)
`System.Security`	Предоставляет классы и интерфейсы для обеспечения безопасности приложений, а также позволяет устанавливать права доступа к различным частям приложения
`System.Web.Secutity`	Содержит классы, обеспечивающие безопасность Web-приложений, которые используют ASP.NET
`System.NET`	Содержит классы, обеспечивающие работу с различными сетевыми протоколами
`System.Configuration`	Предоставляет классы, позволяющие получить программный доступ к настройкам конфигурации .NET Framework
`System.Configuration.Assemblies`	Включает классы, обеспечивающие настройку сборок
`System.Configuration.Install`	Набор классов, служащих для написания инсталляторов для собственных компонентов

Таблица 3.1 *(окончание)*

Пространство имен	Краткое описание
`System.Globalization`	Предоставляет классы, позволяющие создавать многоязычные приложения, с настройками для различных стран (язык, используемый календарь, форматы даты и валюты и многое другое)
`System.Resources`	Содержит классы и интерфейсы для создания, хранения и управления различными ресурсами для локализации приложений
`System.Reflection`	Включает в себя набор классов и интерфейсов для динамического связывания и просмотра типов (а также просмотра загруженных методов и полей)

Таким образом, если вы знаете, как применяется тот или иной класс из того или иного пространства имен в одном языке программирования .NET, вы можете успешно использовать данный класс и в любом другом языке программирования, относящемся к семейству .NET.

Каждый тип в среде .NET представляет собой объект, который прямо или косвенно наследуется от основного класса `Object`.

Примечание

Если при объявлении нового класса вы не укажете базовый класс, то компилятор укажет в качестве такового класс `Object` при генерации IL-кода.

Класс `Object` предоставляет "опубликованные" (`public`) методы, которые наследуются любым другим классом в среде .NET. Эти методы приведены в табл. 3.2.

Таблица 3.2. *"Опубликованные" методы класса* `Object`

Метод	Краткое описание
`Equals()`	Сравнивает два экземпляра класса между собой и определяет их эквивалентность. Если содержимое объектов полностью совпадает, то возвращается значение `True`, в противном случае — `False`
`GetHashCode()`	Предназначен для получения хэш-кода объекта. Отметим, что в среде .NET хэш-коды служат также для определения уникальности объектов во время выполнения приложения
`GetType()`	Служит для получения типа объекта во время выполнения приложения. Результат — тип объекта

Таблица 3.2 (окончание)

Метод	Краткое описание
`ReferenceEquals()`	Сравнивает две ссылки на объект и проверяет, ссылаются ли они на один и тот же объект. Результат — значение `True` (ссылаются на тот же объект) или `False`
`ToString()`	Возвращает строковое представление объекта. По умолчанию результатом является имя класса

Попробуем применить эти методы в простом консольном приложении, написанном на C# (листинг 3.1).

Листинг 3.1. Применение опубликованных методов класса `Object`

```
using System;
namespace MyApplication1
{
   class Class1
   {
     [STAThread]
     static void Main(string[] args)
     {
       //  создаем новый экземпляр (объект) класса Class1
       Class1 c = new Class1();
       //  проверяем, является ли объект эквивалентным самому себе
       Console.WriteLine("Класс эквивалентен — и это:\t" + c.Equals(c));
       //  определяем хэш-код для нашего объекта
       Console.WriteLine("Хэш-код:\t" + c.GetHashCode());
       //  получаем имя объекта
       Console.WriteLine("Имя объекта:\t" + c.ToString());
     }
   }
}
```

Если вы не слишком хорошо знакомы со средой Visual Studio .NET, то отмечу, что для написания приведенного кода можно нажать кнопку **New Project** на стартовой странице **Start Page**. Откроется диалоговое окно **New Project** (рис. 3.1). Выберите среди типов проекта **Project Types** раздел **Visual C# Projects**, а среди шаблонов **Templates** — пиктограмму **Console Application**.

Назовите как угодно ваше приложение (введите в поле **Name** произвольное название), например, `MyApplication1`. Нажмите кнопку **OK** и в открывшемся

окне редактора кода вы увидите заготовку класса `Class1.cs`. Именно в ней записываем код листинга 3.1.

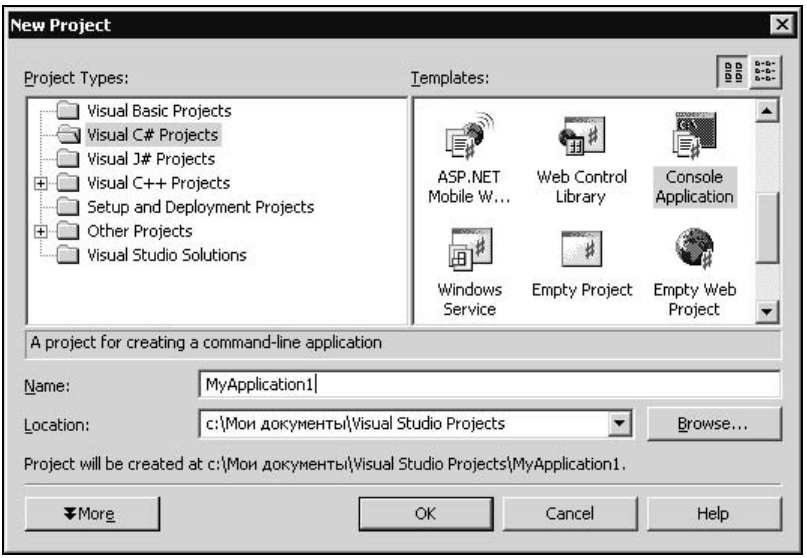


Рис. 3.1. Диалоговое окно **New Project**

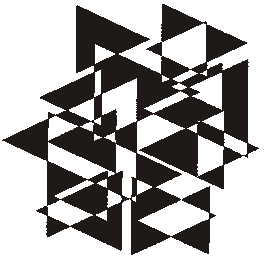
В результате выполнения программы вы увидите следующее:

```
Класс эквивалентен — и это: True
Хэш-код: 2
Имя объекта: MyApplication1.Class1
```

Таким образом, мы рассмотрели общую модель программирования в среде .NET и познакомились с основными пространствами имен .NET Framework.

В *гл. 4* мы опишем работу с .NET-компонентами.

ГЛАВА 4



Работа с распределенными .NET-компонентами

В этой главе кратко, на небольших примерах, рассматриваются распределенные компоненты.

Напомню, что все сборки (приложения, созданные в среде .NET) являются компонентами, которые могут быть подключены к другим компонентам или приложениям. Наличие исходного текста кода при этом совсем необязательно, т. к. все метаданные .NET-компонентов находятся внутри них самих.

Распределенные компоненты

Компонентная технология подразумевает поддержку распределенных вычислений. Для этого необходимо активизировать и вызывать удаленные службы, а также сервисы, относящиеся к другому домену приложения.

Домен приложения (application domain) — это "облегченный" (виртуальный) процесс. Все домены приложения одного процесса Windows могут использовать общее адресное пространство.

В технологии COM для этого использовался протокол связи *DCOM* (Distributed COM). Данный протокол очень хорошо проявил себя при реализации распределенных вычислений, но есть у него и недостатки: плохая работа с брандмауэрами и программным обеспечением NAT, а также большие затраты на управление "временем жизни", инициализацию протоколов и использование двоичных форматов.

Технология .NET позволяет исключить перечисленные выше недостатки, предоставляя множество различных средств поддержки распределенных вычислений. Одним из средств является использование Remoting API. Он позволяет применять различные каналы связи для распределенных вычислений (TCP, HTTP и др.). Кроме того, он может даже подключать свои специальные каналы связи.

Так как среда разработки Visual Studio .NET является полностью объектно-ориентированной, это значительно облегчает распределенные вычисления.

Рассмотрим простой пример создания распределенного приложения, которое выводит строку `Это сообщение` на экран, при вызове приложением-клиентом метода `SayMessage()`, предоставляемого приложением-сервером.

В листинге 4.1 приведен текст приложения-сервера.

Листинг 4.1. Распределенное приложение-сервер

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class CoMessage : MarshalByRefObject
{
   public static void Main()
   {
      TcpChannel channel = new TcpChannel(4000);
      ChannelServices.RegisterChannel(channel);
      RemotingConfigurstion.RegisterWellKnownServiceType(
      typeof(CoMessage),"MyMessage",
      WellKnownObjectMode.Singleton);
      System.Console.WriteLine("Нажмите Enter для выхода");
      System.Console.ReadLine();
   }
   public void SayMessage()
   {
      Console.WriteLine("Это сообщение");
   }
}
```

Так как мы используем в данном примере канал TCP, в начале программы указываются соответствующие пространства имен.

Класс `CoMessage` является потомком класса `MarshalByRefObject`. Это позволяет ссылаться на данный объект из других доменов приложения, а также из других процессов и компьютеров.

Метод `SayMessage()` является "опубликованным" (`public`), поэтому он доступен из внешних клиентских приложений. Этот метод не содержит в себе ничего лишнего, только вывод сообщения на экран, но его сможет вызвать внешнее приложение-клиент, т. к. функция `Main()` нашего приложения-сервера использует класс `TcpChannel`, передавая ему номер порта, из которого сервер будет ожидать запросы клиентов.

Примечание

Для использования протокола HTTP, вместо TCP, достаточно заменить в программе слово `TcpChannel` на `HttpChannel`.

После создания объекта `channel` мы регистрируем его в службе `ChannelServices`, которая обеспечивает регистрацию каналов и разрешения имен объектов. Далее, мы регистрируем объект `CoMessage` с помощью метода `RegisterWellKnownServiceType()` класса `RemotingConfiguration`. Это необходимо для активизации объекта. В качестве параметров данному методу передаются три значения:

- имя класса;
- универсальный идентификатор ресурса (URI, Unique Resource Identifier);
- режим активизации объекта:
  - `Singleton` — один объект обеспечивает множество вызовов;
  - `SingleCall` — один объект обслуживает только один вызов.

Теперь рассмотрим клиентское приложение, текст которого приведен в листинге 4.2.

Листинг 4.2. Приложение-клиент

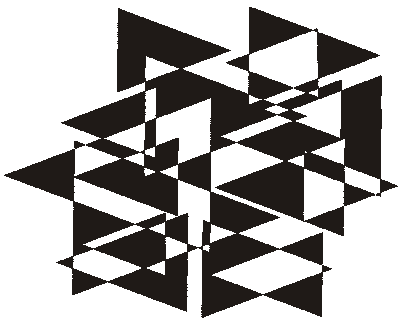
```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class MyClient
{
   public static void Main()
   {
      try
      {
         TcpChannel channel = new TcpChannel();
         ChannelServices.RegisterChannel(channel);
         CoMessage h = (CoMessage) Activator.GetObject(
         typeof(CoMessage),
         "tcp://192.168.1.1:4000/MyMessage");
         h.SayMessage();
      }
      catch(Exception e)
      {
         Console.WriteLine(e.ToString());
      }
   }
}
```

Так как наш класс не является сервером, то он не обязан быть потомком какого-либо конкретного класса.

Для подключения удаленных методов необходимо произвести активизацию объекта. Для этого вызывается метод `GetObject()` класса `Activator`. При этом в качестве параметров передаются имя удаленного класса и его расположение, включая универсальный идентификатор ресурса.

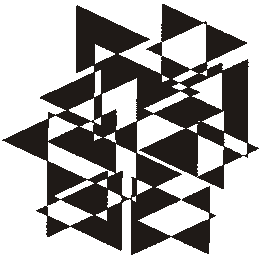
На этом заканчивается вводная *часть I*, описывающая особенности платформы .NET. В *части II* рассказывается о самой визуальной среде разработки Visual Studio .NET 2003, о ее настройках и расширениях.

ЧАСТЬ II



Среда разработки Visual Studio .NET 2003

В этой части речь пойдет о визуальной среде разработки приложений. Описывается механизм настройки данной среды. Вы узнаете, как можно самостоятельно ее расширить. Кроме того, рассказывается, как работать с проектами и решениями, с редактором кода, а также описаны принципы работы с файлами ресурсов.



ГЛАВА 5

Среда разработки и ее настройка

В этой главе познакомимся с основными окнами среды Visual Studio .NET 2003 и принципами их настройки. Кроме того, в конце данной главы мы изучим принципы работы с макросами Visual Studio .NET.

Описание и назначение основных окон Visual Studio .NET 2003

Среда разработки может произвольно настраиваться пользователем. Допустимо настроить ее, например, для языков программирования Visual Basic 6 или Visual C++.

При запуске среды Visual Studio .NET 2003 на экране появится диалоговое окно среды разработки (рис. 5.1). Выберите на открывшейся диалоговой странице **Start Page** вкладку **My Profile**, затем в раскрывающемся списке **Profile** укажите строку **Visual Studio Developer**, если хотите использовать настройки (профайл) для программирования в Visual Studio (без привязки к какому-либо конкретному языку программирования).

После этого в раскрывающихся списках **Keyboard Scheme**, **Window Layout** и **Help Filter** вы можете установить те значения, которые вам необходимы.

Вы всегда можете изменить эти параметры вручную (выбрав нужный из раскрывающегося списка). Диалоговая страница среды разработки доступна в любой момент времени, для ее вызова выберите пункт главного меню **Help | Show Start Page**. Для установки нужного профайла или настройки текущего выберите на открывшейся странице вкладку **My Profile** (так же, как вы уже это сделали ранее).

В большинстве случаев создание новой программы начинается с команды меню **File | New Project**. При этом появляется диалоговое окно создания нового проекта (рис. 5.2).

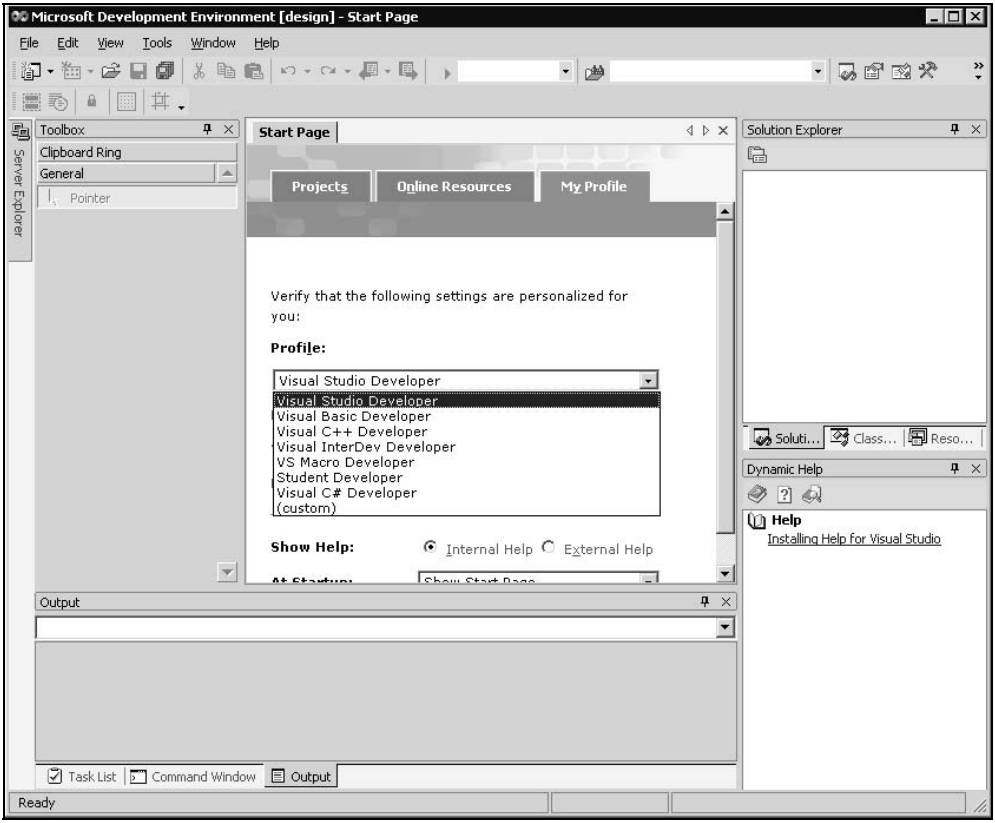


Рис. 5.1. Диалоговое окно среды Visual Studio .NET 2003

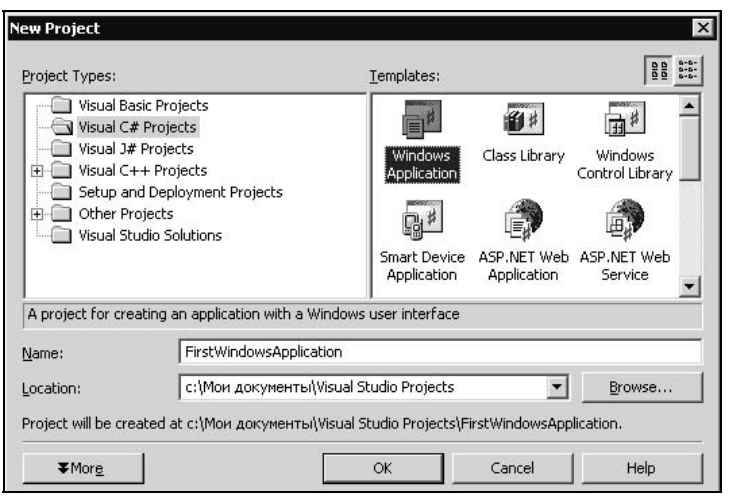


Рис. 5.2. Диалоговое окно создания нового проекта

Попробуем создать первое пробное Windows-приложение на языке C#. В поле **Name** диалогового окна **New Project** (см. рис. 5.2) введем имя нашего проекта `FirstWindowsApplication`. В поле **Location** указывается, где будут размещаться файлы вновь создаваемого проекта. Вы можете изменить путь, записанный в этом поле.

В поле **Project Types** перечислены типы проектов, которые могут быть созданы с помощью данной среды разработки. Так как нам нужно создать проект на C#, выбираем строку **Visual C# Projects**. В поле **Templates** представлены пиктограммы, обозначающие шаблоны, помогающие создать приложения различных видов: обычное Windows-приложение, консольное приложение, библиотеку классов и т. д. Выберем здесь пиктограмму **Windows Application** (Приложение для Windows).

После нажатия кнопки **OK** среда Visual Studio .NET автоматически создает новый проект, который пока состоит из одной формы (рис. 5.3).

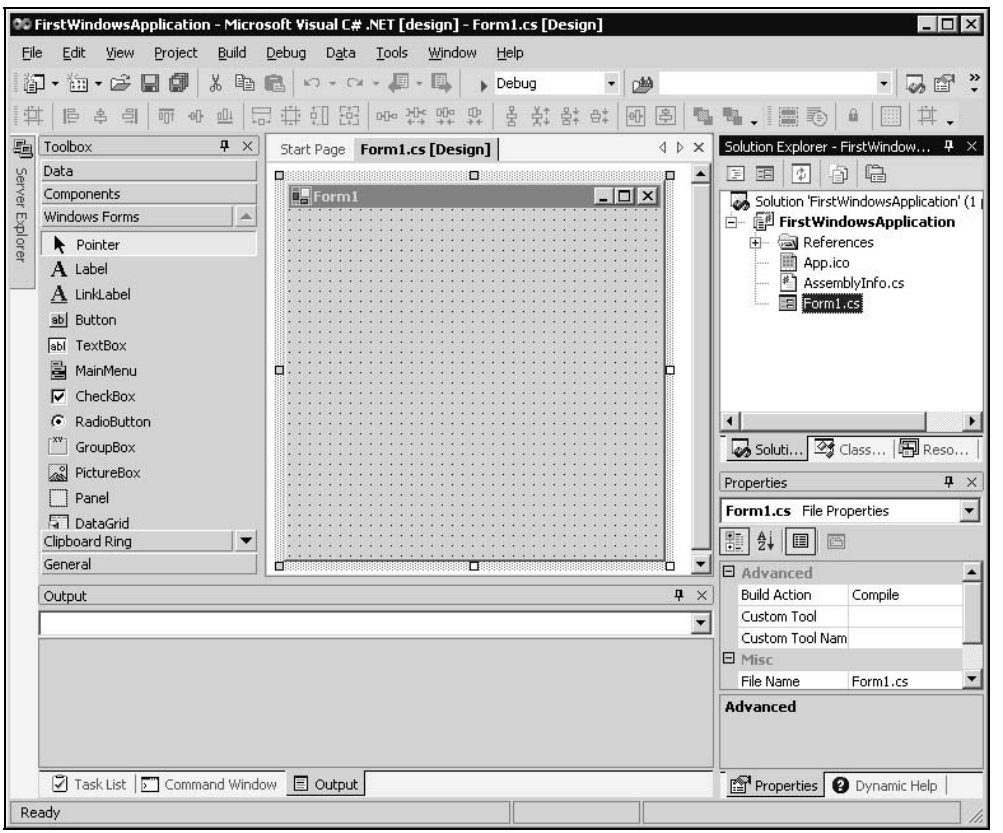


Рис. 5.3. Пустая форма нового приложения для Windows

В принципе, среда разработки Visual Studio .NET уже создала готовое приложение — это обычное окно Windows. Вы можете запустить приложение с помощью клавиши <F5> или при выборе пункта главного меню **Debug | Start**.

Результатом выполнения созданного приложения будет отображение обычного окна Windows (рис. 5.4). Окно можно перемещать на экране, развернуть, свернуть на панель задач. Таким образом, над этим окном можно производить любые действия, применимые к окнам Windows.

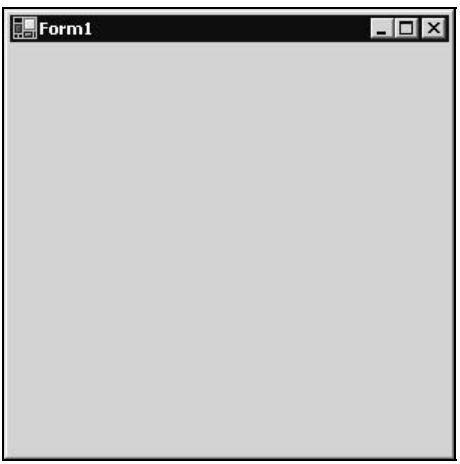


Рис. 5.4. Результат работы нового приложения

Рассмотрим теперь основные окна среды разработки. Именно для этого нам и понадобилось создать простое приложение.

Окно редактора

Окно редактора показано на рис. 5.3 (область, где располагается заготовка формы **Form1**), а также на рис. 5.5 (область текста `using System`).

Редактор Visual Studio .NET 2003 позволяет выполнять все стандартные действия, которые доступны для любого другого редактора (копирование, вставка, поиск и т. д.).

Чтобы настроить параметры окна редактора, вы можете выбрать пункт главного меню **Tools | Options**. При этом появится диалоговое окно настройки параметров среды (рис. 5.6).

В левой части диалогового окна откроем папку **Text Editor** и вложенную папку **General**. После чего, в правой части диалогового окна появятся основные параметры окна редактора. Для настройки параметров редактора языка C# выберите **Text Editor | C#**.

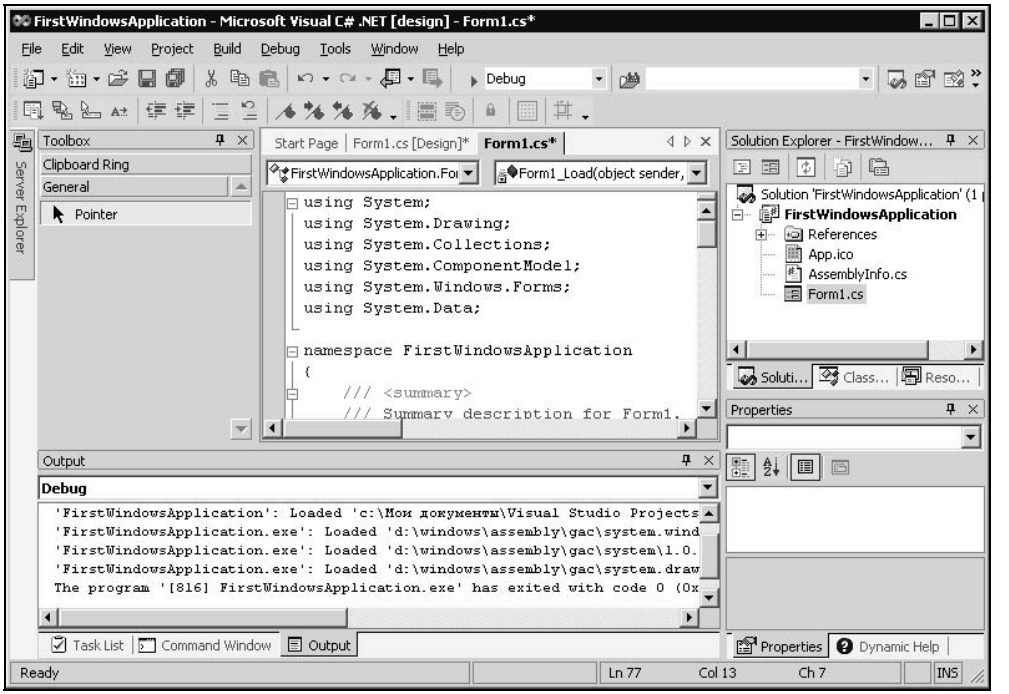


Рис. 5.5. Окно редактора внутри оболочки среды разработки

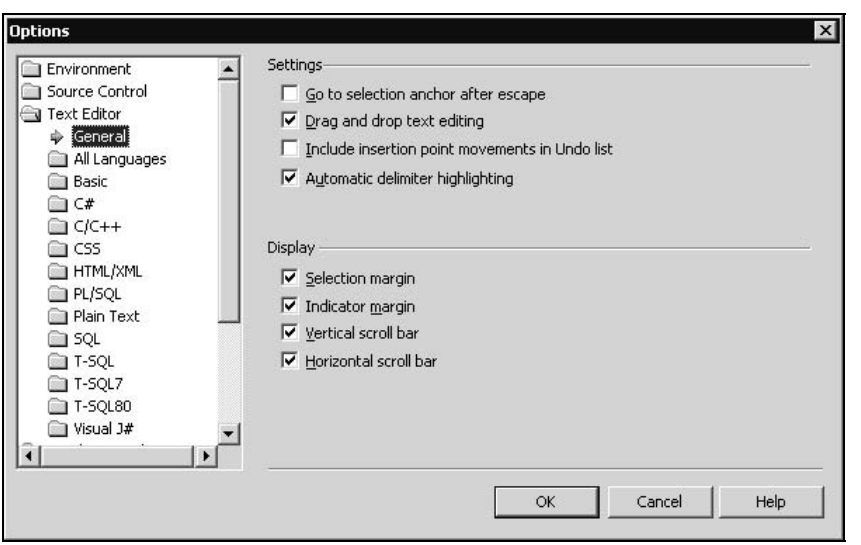


Рис. 5.6. Диалоговое окно **Options**

Рассмотрим общие параметры окна редактора (см. рис. 5.6):

- область **Settings** (Установки):
  - флажок **Go to selection anchor after escape** — если установлен, оставляет курсор в том месте текста программы, откуда вы начали выделение, а затем нажали клавишу <Esc> для отмены выделения. Например, если вы начали выделять текст с конца строки до начала строки, то при отмене выделения, с помощью клавиши <Esc>, курсор будет находиться в конце строки;
  - флажок **Drag and drop text editing** — если установлен, то позволяет перемещать выделенный текст в любую часть программы, путем его перетаскивания (метод `Drag And Drop`);
  - флажок **Include insertion point movements in Undo list** — если установлен, то начальные точки нахождения перемещаемых команд будут запоминаться в списке **Undo** (Отмена);
  - флажок **Automatic delimiter highlighting** — если установлен, то символы, разделяющие инструкции, будут подсвечиваться;
- область **Display** (Отображение):
  - флажок **Selection margin** — если установлен, то поле выбора слева от текста программы будет отображаться. Это поле обведено на рис. 5.7;

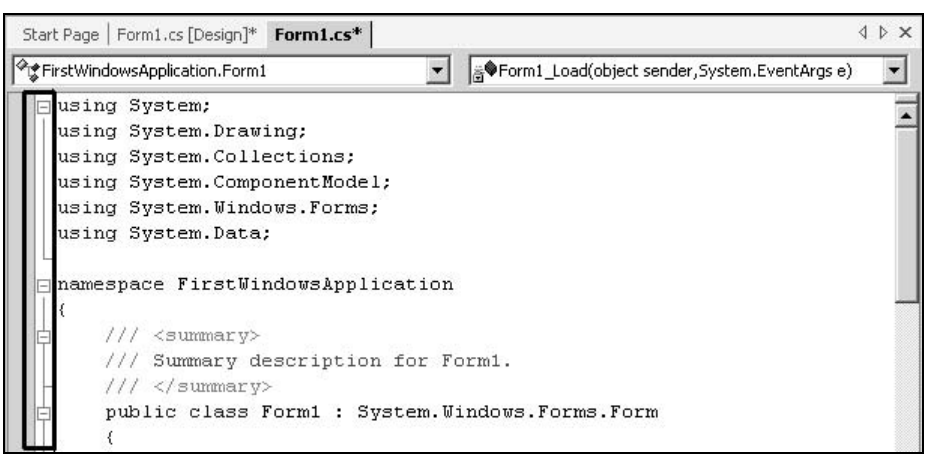


Рис. 5.7. Окно редактора с выделенным (черным прямоугольником) полем выбора

- флажок **Indicator margin** — если установлен, то серое поле слева от окна редактора будет отображаться — данное поле используется, например, для установки точек прерываний, речь о которых пойдет в *гл. 9*;

- флажок **Vertical scroll bar** — если установлен, то в правой части окна редактора кода будет отображаться вертикальная полоса прокрутки. В противном случае, полосы прокрутки не будет, а для перемещения по тексту программы вверх и вниз вы можете использовать клавиши <Page Up>, <Page Down> и клавиши управления курсором;
- флажок **Horizontal scroll bar** — если установлен, то в нижней части окна редактора кода будет отображаться горизонтальная полоса прокрутки. Если флажок снять, полосы прокрутки не будет, а для перемещения по тексту программы влево и вправо вы можете использовать клавиши управления курсором.

Переходим к рассмотрению общих параметров редактора для всех языков программирования (рис. 5.8, правая часть). Для этого в левой части диалогового окна **Options** (см. рис. 5.6) выберите папку **Text Editor** и вложенную папку **All Languages | General**.

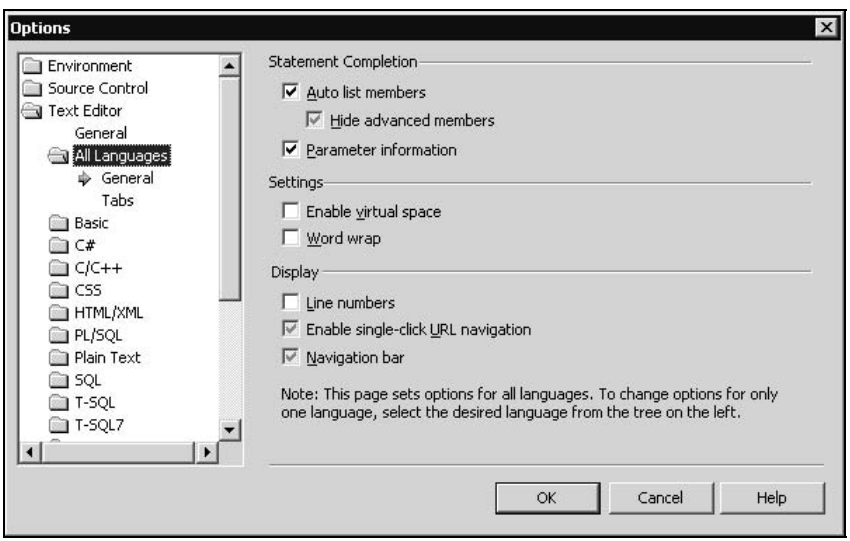


Рис. 5.8. Основные параметры окна редактора для всех языков программирования

Рассмотрим эти параметры подробно:

- ❒ область **Statement Completion** (Завершение кода):
  - флажок **Auto list members** — если установлен, то при вводе имени класса или объекта, после нажатия точки будет отображаться список доступных свойств и методов данного класса или объекта, в виде списка;
  - флажок **Hide advanced members** — если установлен, то недоступные вне класса свойства и методы не будут отображаться после нажатия точки, сразу после имени класса;

- флажок **Parameter information** — если установлен, то при обращении к процедуре или функции, сразу после того, как вы напишите имя и откроете круглую скобку, будет показан список параметров данной функции или процедуры;

❒ область **Settings** (Установки):

- флажок **Enable virtual space** — если установлен, то вы можете перемещать курсор за пределы конца строки. Если вы начнете печатать текст за пределами конца строки, промежуток между концом строки и вновь набираемым текстом автоматически заполнится пробелами. Если же флажок не установлен, то переместить курсор за пределы строки будет невозможно;
- флажок **Word wrap** — если установлен, то текст, выходящий справа за пределы окна редактора, будет автоматически перенесен на строку ниже. Таким образом, весь текст будет отображаться в пределах окна редактора кода;

❒ область **Display** (Отображение):

- флажок **Line numbers** — если установлен, то слева от каждой строки текста программы будет показан номер этой строки (как в не визуальном языке Basic);
- флажок **Enable single-click URL navigation** — если установлен, то адрес в сети Интернет (URL), находящийся в тексте программы, позволит вам перейти на Web-страницу простым щелчком левой кнопкой мыши по нему;
- флажок **Navigation bar** — данный флажок не доступен в редакторе Visual Basic.

Рассмотрим теперь настройки параметров табуляций и отступов окна редактора для всех языков программирования (рис. 5.9, правая часть). Для этого в левой части диалогового окна **Options** (см. рис. 5.6) выберите папку **Text Editor** и вложенную папку **All Languages | Tabs**.

Опишем эти параметры:

❒ область **Indenting** (Отступ):

- переключатель **None** — если включен, то при переходе на следующую строку с помощью клавиши <Enter> отступы в левой части строки не будут добавлены. Курсор поместится на первую левую позицию новой строки;
- переключатель **Block** — если включен, то при переходе на следующую строку с помощью клавиши <Enter> будут добавлены такие же отступы слева, как и в предыдущей строке;
- переключатель **Smart** — если включен, то новые строки текста будут автоматически форматироваться по правилам отступов для текущего языка;

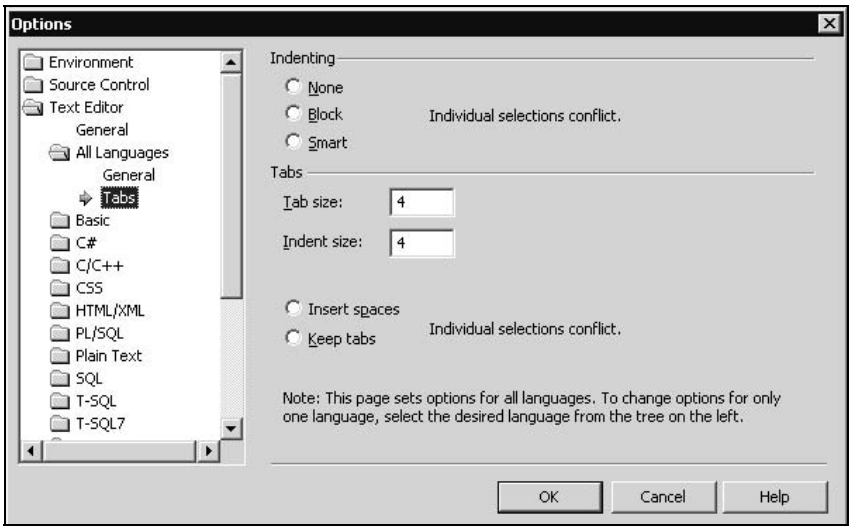


Рис. 5.9. Параметры табуляций окна редактора для всех языков программирования

- область **Tabs** (Табуляция):
  - поле **Tab size** — определяет количество пробелов, равносильных символу табуляции;
  - поле **Indent size** — определяет количество пробелов, вводимых при нажатии клавиши <Tab>;
  - переключатель **Insert spaces** — если включен, то при нажатии клавиши <Tab> будут вводиться символы пробела. Например, при установке значений **Tab size** и **Indent size** равных 4, при нажатии клавиши <Tab> будет введено 4 пробела вместо одного символа табуляции;
  - переключатель **Keep tabs** — если включен, то при нажатии клавиши <Tab> будут вводиться символы табуляции.

Точно так же выполняются настройки редактора кода для любого конкретного языка. Его нужно лишь выбрать в левой части диалогового окна **Options** (см. рис. 5.6). Более подробно работу с редактором кода мы рассмотрим в *гл. 8*.

Окно просмотра решения

Любой проект, который вы создаете в Visual Basic .NET, является частью *решения* (solution). В *окне просмотра решения* (**Solution Explorer**) отображается список всех файлов, которые входят в текущее решение (рис. 5.10).

Это окно находится справа от окна редактора кода (см. рис. 5.5). К любому из перечисленных файлов решения вы можете быстро перейти, дважды щелкнув по нему в окне просмотра решения левой кнопкой мыши.

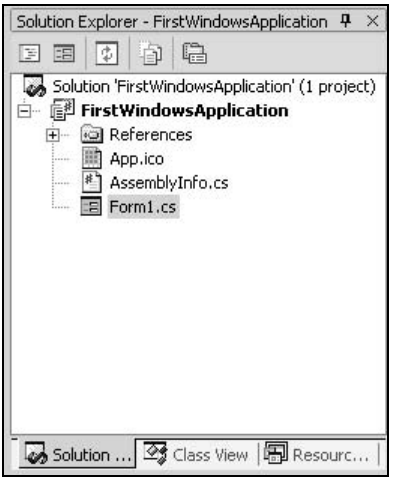


Рис. 5.10. Окно просмотра решения (**Solution Explorer**)

Окно свойств

С помощью *окна свойств* (рис. 5.11), которое расположено под окном просмотра решения (см. рис. 5.10), вы можете быстро получить доступ к свойствам любых компонентов и элементов управления.

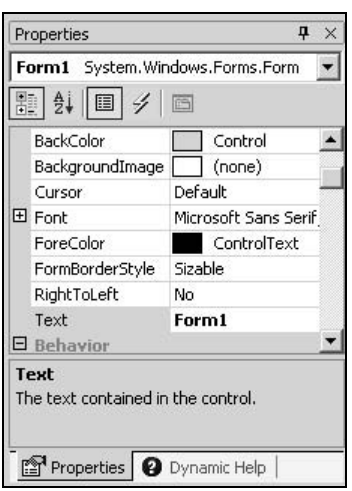


Рис. 5.11. Окно свойств

Окно свойств содержит свойства формы `Form1`. Вы можете произвольно менять значения этих свойств.

В верхней части окна имеется раскрывающийся список, который позволяет выбрать объект, свойства которого будут отображаться в данном окне (в нашем случае, это `Form1`).

На панели, расположенной ниже раскрывающегося списка, имеется пять пиктограмм, назначение которых следующее (описаны слева направо):

Categorized — позволяет отображать свойства разбитыми на категории. Выключается путем включения следующей пиктограммы;

Alphabetic — позволяет отображать свойства в алфавитном порядке без разбивки на категории. Отключается путем включения предыдущей пиктограммы;

Properties — показывает, что в настоящий момент времени окно отображает свойства какого-либо объекта;

Events — показывает, что в настоящий момент времени окно отображает события какого-либо объекта;

Property Pages — позволяет отображать страницы свойств, если они существуют (страницы свойств облегчают ввод сложных значений некоторых свойств).

В нижней части окна вы можете видеть краткое описание того свойства, которое в настоящий момент выделено (в нашем примере, это свойство **Text**).

Окно вывода и окно команд

В *окне вывода* (**Output**) (рис. 5.12), которое можно вызвать с помощью комбинации клавиш <Ctrl>+<Alt>+<O> отображается текущая информация о состоянии вашего решения. Это окно располагается ниже окна редактора кода (см. рис. 5.5).

При компиляции программы в окне вывода будет отображаться информация об успешной компиляции или об ошибках, возникших в процессе компиляции.

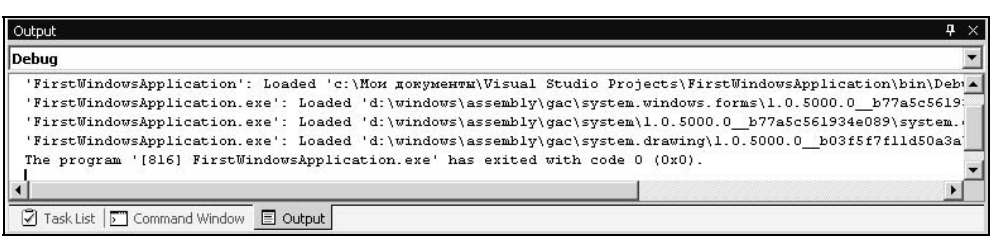


Рис. 5.12. Окно вывода

Окно команд (**Command Window**) (рис. 5.13) можно вызвать с помощью комбинации клавиш <Ctrl>+<Alt>+<A>. Данное окно применяется в процессе отладки ваших приложений.

В него можно вписать любую команду, которая будет мгновенно исполнена.

Рис. 5.13. Окно команд

Настройки среды

Чтобы настроить параметры среды разработки Visual Studio .NET 2003, необходимо выбрать пункт главного меню **Tools | Options**. При этом появится диалоговое окно настройки параметров среды. В левой части этого окна выберем папку **Environment** и вложенную папку **General**. В результате, мы увидим общие параметры среды (рис. 5.14, правая часть).

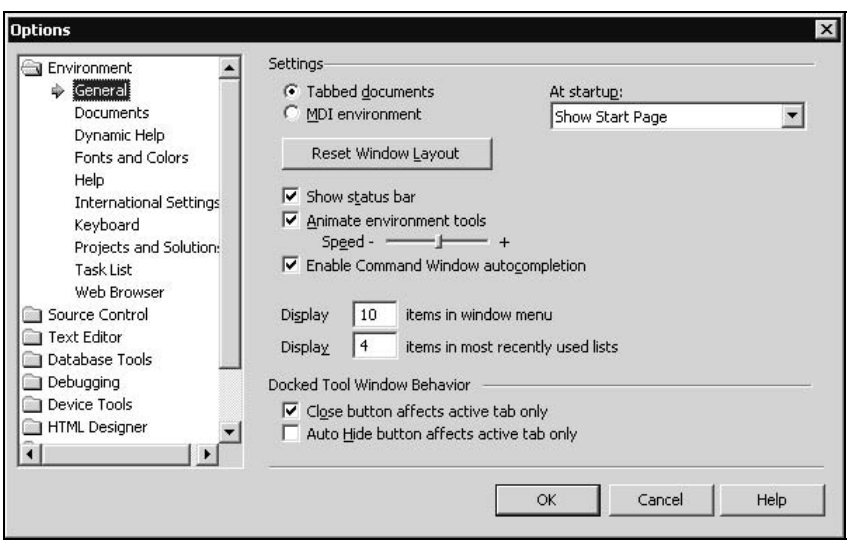


Рис. 5.14. Общие настройки среды разработки

Рассмотрим некоторые из них подробно:

- область **Settings** (Установки):
  - переключатель **Tabbed documents** — если включен, то все файлы, открываемые в окне редактора кода, будут расположены на различных вкладках одного окна. Если режим был переключен из следующего (**MDI environment**), то необходимо перезапустить среду разработки;
  - переключатель **MDI environment** — если включен, то каждый из файлов окна редактора кода будет отображаться отдельно от остальных. Для переключения между файлами вы можете использовать комбинацию клавиш <Ctrl>+<Tab>. Если режим был переключен из предыдущего (**Tabbed documents**), то необходимо перезапустить среду разработки;
  - раскрывающийся список **At startup** — позволяет выбрать, что будет загружаться и показываться при каждом начальном запуске среды разработки. Варианты: **Show Start Page** — будет отображена стартовая страница, **Load last loaded solution** — будет загружено последнее разрабатываемое вами решение (solution), **Show Open Project dialog box** — будет отображено диалоговое окно открытия проекта, **Show New Project dialog box** — будет отображено диалоговое окно создания нового проекта, **Show empty environment** — будет загружена только оболочка среды;
  - кнопка **Reset Window Layout** — позволяет сбросить все измененные вами настройки в такие, которые приняты по умолчанию в среде Visual Studio .NET 2003;
  - флажок **Show status bar** — если установлен, то в нижней части окна среды разработки будет отображаться панель состояния с различной служебной информацией о происходящих операциях;
  - флажок **Animate environment tools** — если установлен, то при открытии и закрытии автоматически закрываемых окон будут использоваться эффекты переходов;
  - бегунок **Speed** — позволяет установить скорость выполнения переходов, если флажок **Animate environment tools** установлен;
  - флажок **Enable Command Window autocompletion** — если установлен, то в командном окне будет использоваться механизм автоматического завершения команд;
  - поле **Display ... items in window menu** — позволяет установить количество окон, которые будут отображаться в меню **Window** главного меню среды разработки, для быстрого перехода от одного окна к другому. Может принимать значения от 1 до 24, по умолчанию имеет значение 10;

- поле **Display ... items in most recently used lists** — позволяет установить число недавно использовавшихся вами проектов, которые отображены в меню **File** главного меню среды разработки. Это позволяет быстро вернуться к нужному проекту. Может принимать значения от 1 до 24, по умолчанию имеет значение 4.

Выберем теперь в левой части диалогового окна настроек папку **Environment** и вложенную папку **Documents** (рис. 5.15). В результате, в правой части окна мы увидим параметры работы среды с документами.

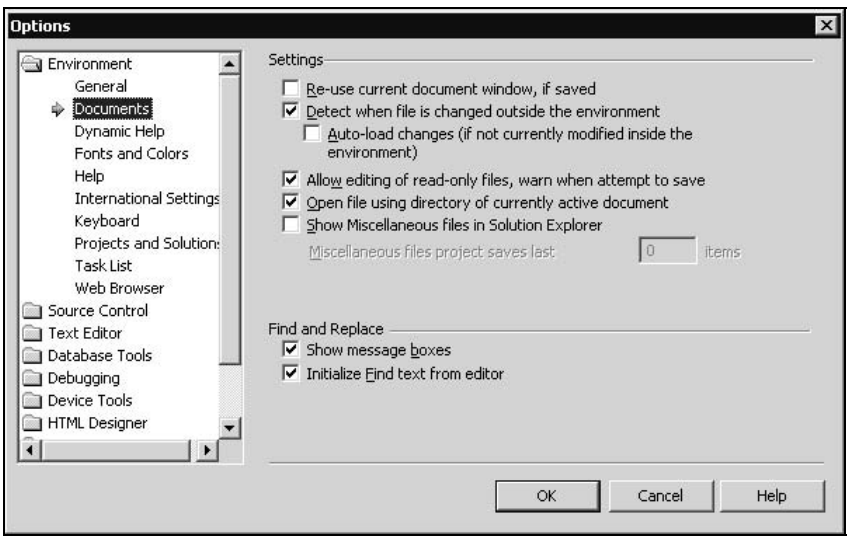


Рис. 5.15. Параметры работы среды с документами

Опишем эти параметры подробно:

❒ область **Settings** (Установки):

- флажок **Re-use current document window, if saved** — если установлен, то при открытии нового документа, если текущий документ был сохранен, он будет открыт в том же окне, что и текущий. Если текущий документ не был сохранен, или если данный флажок не установлен, новый документ будет открываться в другом окне;
- флажок **Detect when file is changed outside the environment** — если установлен, то при открытии файла, который был изменен вне среды разработки, будет выдаваться сообщение об этом. Кроме того, вы сможете загрузить прежний, не измененный файл;
- флажок **Auto-load changes (if not currently modified inside the environment)** — если предыдущий и данный флажки установлены, то сооб-

щение о том, что файл был изменен вне среды разработки, выдаваться не будет, просто все изменения будут приняты средой разработки как внутренние;

- флажок **Allow editing of read-only files, warn when attempt to save** — если установлен, то вы сможете открывать и редактировать файлы, помеченные, как "только для чтения" (**Read-only**). После редактирования вы сможете сохранить их под другим именем, используя пункт главного меню среды разработки **File | Save As**;
- флажок **Open file using directory of currently active document** — если установлен, то диалоговое окно открытия файла будет показывать содержимое каталога, в котором находится текущий документ. В противном случае (если флажок не установлен), будет отображаться содержимое каталога, который был использован в последний раз для открытия файла;
- флажок **Show Miscellaneous files in Solution Explorer** — если установлен, то в окне просмотра решения появится раздел, в котором будут отображаться различные файлы, не связанные с проектом или решением, но, по каким-либо причинам, включенные вами в это окно;
- поле **Miscellaneous files project saves last ... items** — определяет, сколько недавно использовавшихся вами различных файлов, не связанных с проектом или решением, должно отображаться в окне просмотра решения. Может принимать значения от 0 до 256. По умолчанию имеет значение 0;

❒ область **Find and Replace** (Поиск и замена):

- флажок **Show message boxes** — если установлен, то во время операций поиска и замены текста будут отображаться окна сообщений с надписью "Don't show me again" (не показывать снова), например, если текст не был найден. Если вы выключите этот флажок или включите флажок **Don't show me again** в появившемся окне, окна сообщений появляться не будут;
- флажок **Initialize Find text from editor** — если установлен, то в окне поиска и замены текста будет автоматически вставляться текст, который находился вблизи курсора в момент вызова окна поиска и замены. Окно поиска и замены вызывается комбинацией клавиш <Ctrl>+<F> или с помощью пункта главного меню **Edit | Find and Replace** среды разработки.

В левой части диалогового окна настроек выберем папку **Environment** и вложенную папку **Fonts and Colors**. В правой части диалогового окна (рис. 5.16) отобразятся параметры используемых в среде разработки шрифтов и цветовых схем.

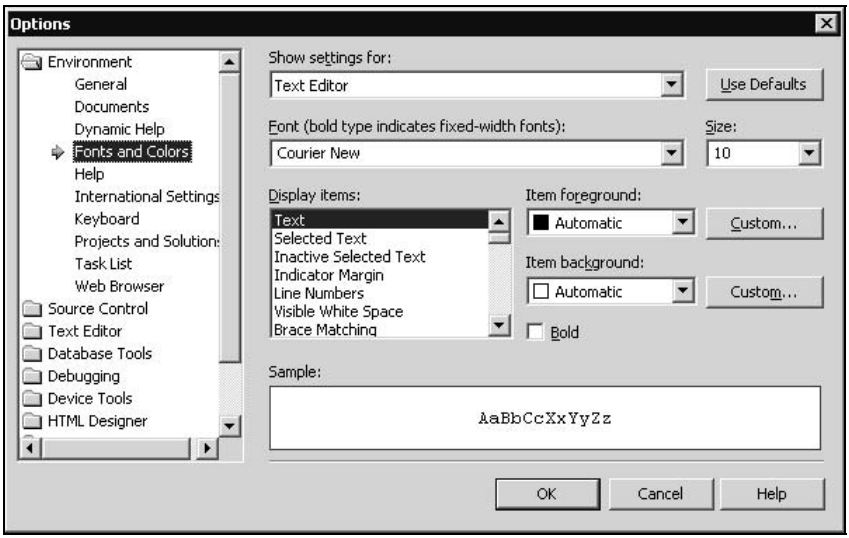


Рис. 5.16. Настройки шрифтов и цветовых схем среды разработки

Рассмотрим их подробно:

- раскрывающийся список **Show settings for** — содержит список всех элементов интерфейса пользователя, которые используются в среде разработки Visual Studio .NET 2003. Если вы выберете любой элемент из данного списка, то затем сможете настроить шрифт и цвет составляющих данного элемента, выбрав их в списке **Display items**;
- кнопка **Use Defaults** — осуществляет сброс всех настроек на стандартные, принятые по умолчанию в среде разработки;
- раскрывающийся список **Font** — позволяет выбрать и установить шрифт, который будет использоваться для данного элемента интерфейса;
- раскрывающийся список **Size** — служит для установки размера используемого шрифта;
- список **Display items** — содержит список составляющих элемента интерфейса пользователя среды разработки, для которых можно установить цвет фона и цвет букв;
- раскрывающийся список **Item foreground** — позволяет выбрать основной цвет (цвет букв) для выбранной составляющей элемента интерфейса;
- кнопка **Custom** — открывает диалоговое окно выбора нужного цвета;
- раскрывающийся список **Item background** — позволяет выбрать цвет фона для выбранной составляющей элемента интерфейса;
- кнопка **Custom** — открывает диалоговое окно выбора нужного цвета фона;

- флажок **Bold** — позволяет установить жирный шрифт для данной составляющей элемента интерфейса;
- графическое поле **Sample** — отображает пример текста с выбранными установками.

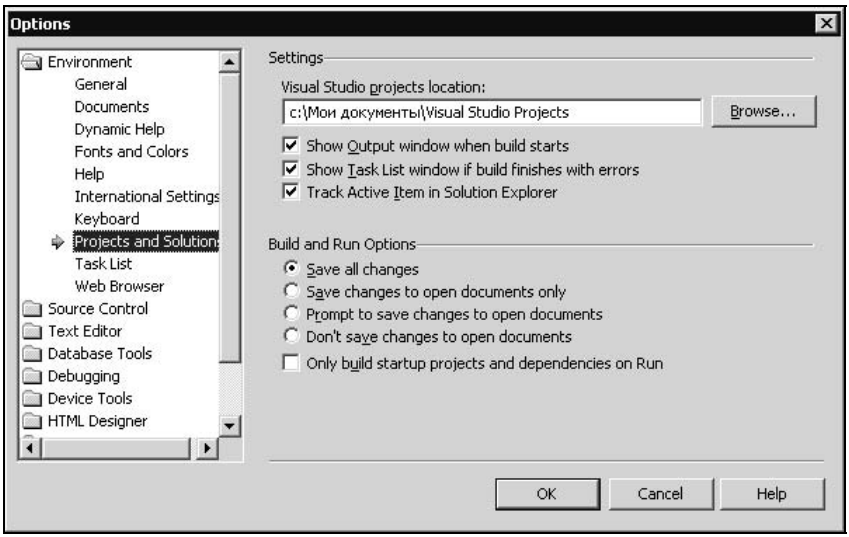


Рис. 5.17. Настройки проектов и решений

Перейдем теперь в диалоговом окне настроек к папке **Environment** и вложенной папке **Projects and Solutions**, расположенным в левой части окна. В правой части диалогового окна находятся параметры, определяющие настройки проектов и решений (рис. 5.17):

- область **Settings** (Установки):
  - поле **Visual Studio projects location** — позволяет установить путь к папке, в которой будут храниться все проекты и решения, создаваемые в среде Visual Studio .NET;
  - кнопка **Browse** — открывает диалоговое окно, в котором вы можете указать нужную папку для хранения проектов и решений, создаваемых в среде Visual Studio .NET;
  - флажок **Show Output window when build starts** — если установлен, то в процессе компиляции проекта автоматически появится окно вывода;
  - флажок **Show Task List window if build finishes with errors** — если установлен и в процессе компиляции проекта возникли ошибки, то среда Visual Studio .NET автоматически отобразит окно списка текущих задач и включит в этот список ошибки компиляции;

- флажок **Track Active Item in Solution Explorer** — если установлен, то при переходе на другой объект, если он имеется в окне **Solution Explorer**, данный объект будет автоматически выделен в окне просмотра решения;

❒ область **Build and Run Options** (Настройки запуска и компиляции):

- переключатель **Save all changes** — если включен, то все изменения во всех файлах решения будут автоматически сохраняться;
- переключатель **Save changes to open documents only** — если включен, то во всех открытых документах все изменения автоматически будут сохраняться при запуске или компиляции проекта, без запроса разрешения на сохранение;
- переключатель **Prompt to save changes to open documents** — если включен, то изменения будут сохраняться, но прежде будет производиться запрос разрешения на это сохранение;
- переключатель **Don't save changes to open documents** — если включен, то изменения в документах, сделанные до компиляции, не будут сохраняться;
- флажок **Only build startup projects and dependencies on Run** — если установлен, то будут компилироваться только те файлы, которые необходимы для запуска приложения.

Работа с макросами Visual Studio

Очень часто при создании приложений возникает необходимость повторять несколько раз одну и ту же последовательность действий. В более ранних версиях Microsoft Visual Studio программистам приходилось повторять такие последовательности действий самостоятельно. В новой версии Visual Studio .NET вы можете автоматизировать выполнение однотипных последовательностей действий с помощью макросов.

Макрос в Visual Studio .NET — это последовательность инструкций (действий), которые могут быть записаны в файл с расширением Vsmacros и затем выполнены в любое время.

Если вы хорошо знаете такие программы, как Microsoft Word и Microsoft Excel, то макросы должны быть вам знакомы.

Макросы могут быть созданы одним из двух способов:

❒ записаны с помощью среды разработки. Вы просто выполняете нужную последовательность действий, а макрос будет записываться автоматически;

❒ записаны с помощью набора кода макроса в специальной среде разработки Macros IDE.

Рассмотрим кратко оба варианта записи макросов.

Для записи макроса с помощью среды разработки необходимо выполнить следующие действия:

1. Выбрать в главном меню среды разработки пункт **Tools | Macros | Record TemporaryMacro** или нажать комбинацию клавиш <Ctrl>+<Shift>+<R>. При этом начнется запись всех выполненных вами действий. Кроме того, появится панель инструментов записи макроса **Recorder** (рис. 5.18). Первая пиктограмма панели позволяет приостановить запись макроса (повторное нажатие возобновляет запись), вторая пиктограмма служит для остановки записи макроса, а третья — отменяет запись макроса.
2. Выполнить все действия, которые должны выполняться автоматически.
3. Остановить запись макроса с помощью панели **Recorder** или комбинацией клавиш <Ctrl>+<Shift>+<R>.

Рис. 5.18. Панель **Recorder**

Записанный таким образом макрос автоматически получит название **TemporaryMacro**. Вы можете переименовать его посредством просмотрщика макросов (**Macro Explorer**), который можно отобразить с помощью комбинации клавиш <Alt>+<F8> или выбором пункта **Tools | Macros | Macro Explorer** из главного меню среды разработки (рис. 5.19).

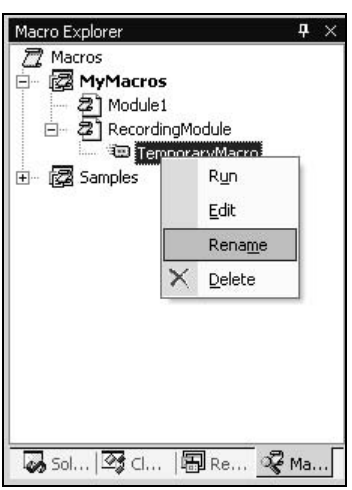


Рис. 5.19. Окно **Macro Explorer**

Запуск записанного макроса можно выполнить тоже с помощью окна **Macro Explorer**. Для этого необходимо в раскрывающемся меню нужного макроса (вызывается щелчком правой кнопкой мыши) выбрать пункт **Run**.

Кроме этого способа вы можете записывать макросы самостоятельно, с использованием специальной среды разработки макросов — Macros IDE. Данная среда позволяет также производить отладку макросов.

Чтобы запустить среду Macros IDE, выберите пункт главного меню Visual Studio .NET 2003 **Tools | Macros | Macros IDE** или используйте комбинацию клавиш <Alt>+<F11>. В окне со средой Macros IDE (рис. 5.20) вы можете просматривать код уже готовых макросов, а также изменять и записывать код для любых макросов.

В окне 5.20 отображается текст макроса, записанного с помощью среды разработки (обычным способом). Этот макрос выполняет вставку в текст числа 123.

Не будем останавливаться на изучении внутреннего строения макросов. Если вас заинтересовала данная тема, обратитесь к справке по Macros IDE.

Итак, в этой главе я рассказал вам об основных окнах и параметрах настройки среды разработки Microsoft Visual Studio .NET 2003, а также привел несколько слов о работе с макросами.

В *гл. 6* будут рассматриваться вопросы дополнения и изменения среды разработки.

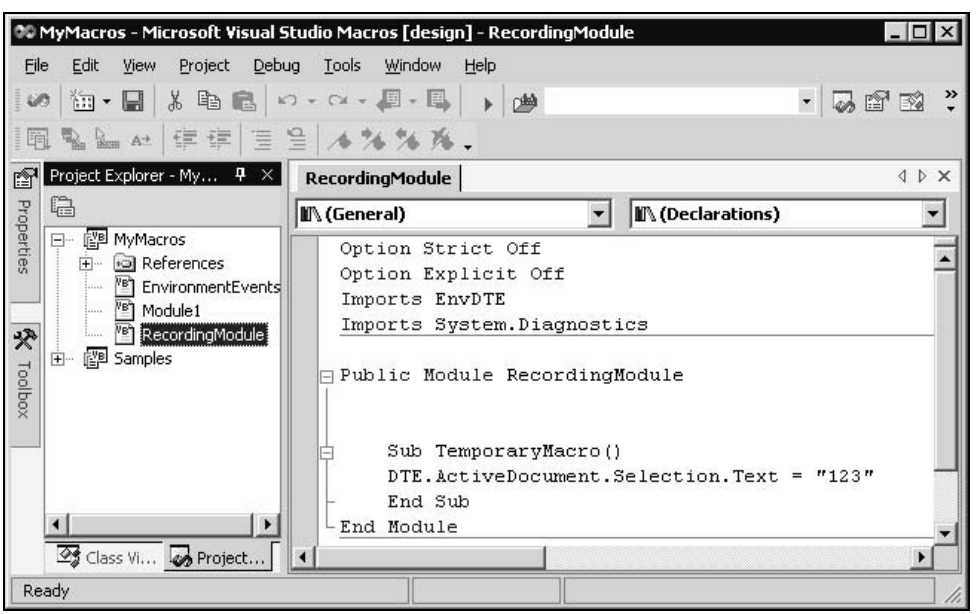
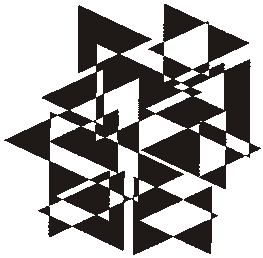


Рис. 5.20. Окно со средой Macros IDE

ГЛАВА 6



Расширение среды разработки

В этой главе мы рассмотрим способы создания своих собственных дополнений к среде разработки Visual Studio .NET 2003. Вы научитесь создавать и изменять окна среды разработки, а также создавать Мастера (Wizards) среды.

Создание дополнений

Среда разработки Visual Studio .NET предоставляет программисту возможность не только настраивать уже готовые средства разработки, но и создавать свои собственные с тем, чтобы встраивать их в интегрированную среду.

Одним из средств расширения среды Visual Studio .NET является создание рассмотренных в *гл. 5* макросов.

Если вы хотите создать новые возможности среды разработки, то вам придется использовать так называемые *расширения* (Add-ins). Расширения позволяют сделать многое из того, чего нельзя достичь, используя макросы. Это такие возможности, как:

- создание своих собственных страниц свойств в диалоговом окне **Options** (вызывается при помощи пункта **Tools | Options** главного меню среды разработки);
- создание собственных панелей инструментов, похожих на стандартные панели инструментов, используемые в среде Visual Studio .NET;
- динамическое включение и выключение пунктов меню;
- добавление различной информации в стандартное диалоговое окно **About** (О программе), которое генерируется для проектов средой разработки.

Для создания расширений используется специальный Мастер расширений (Add-in Wizard). Рассмотрим процесс создания расширения с помощью Мастера.

1. Выберите пункт **File | New Project** в главном меню среды разработки.
2. В появившемся диалоговом окне **New Project** укажите тип проекта в поле **Project Types**, открыв папку **Other Projects** и вложенную папку **Extensibility Projects**. В поле **Templates** выберите шаблон **Visual Studio .NET Add-in**, как показано на рис. 6.1.

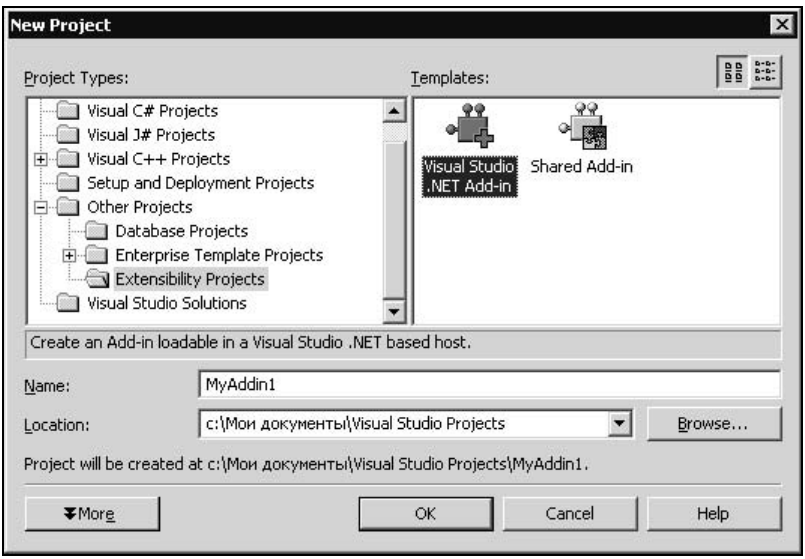


Рис. 6.1. Начало создания расширения Add-in

3. В поле **Name** диалогового окна **New Project** введите имя расширения, например, `MyAddin1`, а в поле **Location** — укажите путь к папке, в которой разместится расширение.
4. После нажатия кнопки **OK** появится стартовое окно Мастера расширений (рис. 6.2).
5. Нажмите кнопку **Next** и перейдите к первому шагу (из шести), который позволяет выбрать язык программирования для создания расширения (рис. 6.3). Вы можете указать один из трех языков: Visual C#, Visual Basic или Visual C++. В нашем примере выбран язык Visual C#. Нажмите кнопку **Next**.
6. На втором шаге Мастера (рис. 6.4) вы можете выбрать те приложения, которые смогут использовать расширение после его создания. По умолчанию, выбраны оба приложения: Microsoft VSMacros IDE и Microsoft Visual Studio .NET. Оставьте все как есть и нажмите кнопку **Next**.
7. Третий шаг Мастера (рис. 6.5) позволяет выбрать имя создаваемого расширения и дать ему краткое описание. Назовем расширение `MyFirstAddin` и дадим ему краткое описание. Затем нажмите кнопку **Next**.

Рис. 6.2. Стартовое окно Мастера расширений

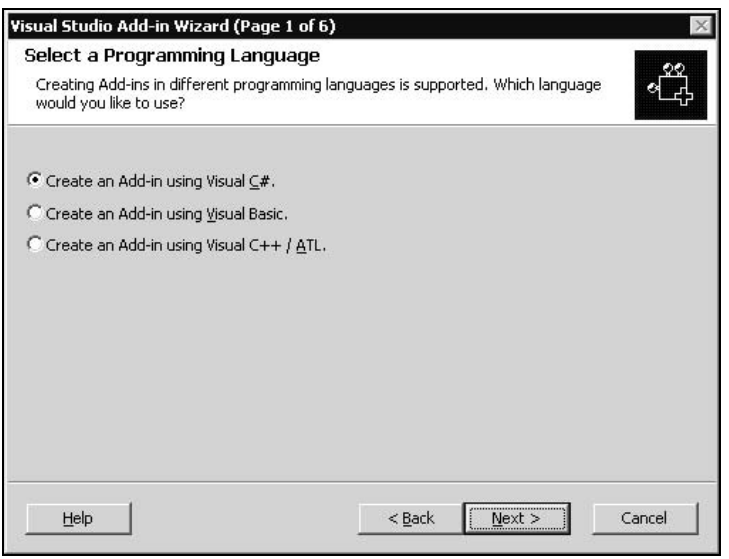


Рис. 6.3. Первый шаг Мастера расширений

8. На четвертом шаге Мастера вы можете указать, должно ли расширение отображаться на панели инструментов, стартовой странице, и другие опции. Оставьте все без изменений и нажмите кнопку **Next**.

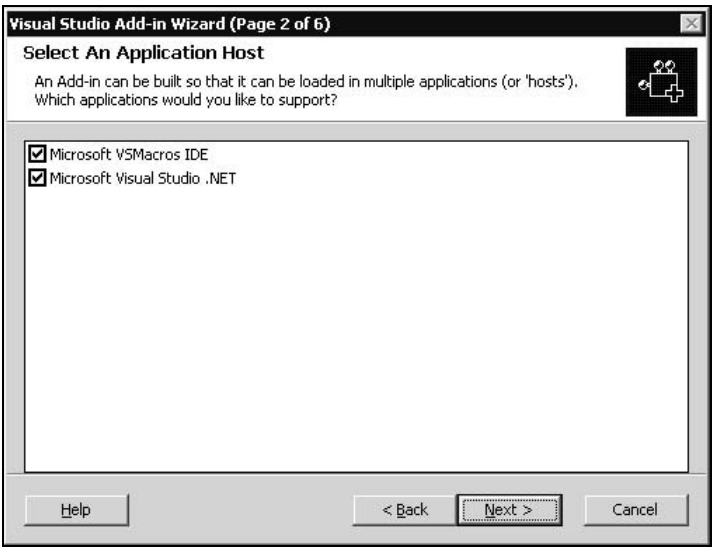


Рис. 6.4. Второй шаг Мастера расширений

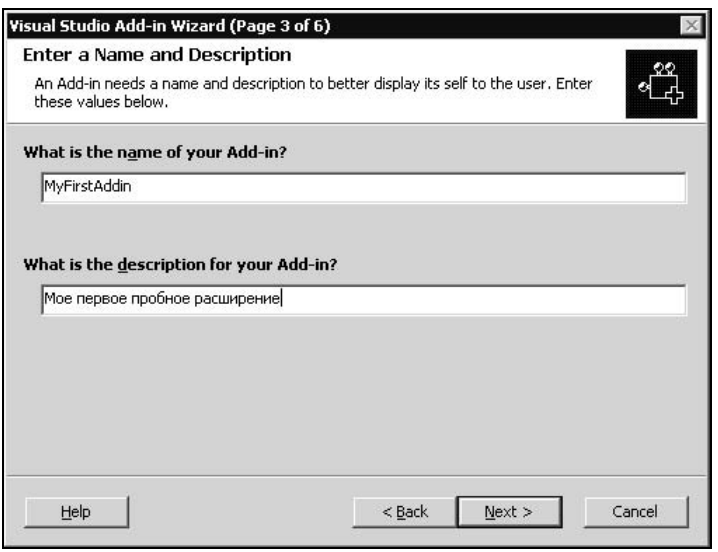


Рис. 6.5. Третий шаг Мастера расширений

9. Пятый шаг Мастера позволяет добавить текстовую информацию в окно **About**, которая автоматически добавится в расширение. Таким образом, вы можете указать сведения о себе, как о создателе данного расширения. Оставьте все без изменений и нажмите кнопку **Next**.

10. Последнее окно Мастера (шестой шаг) отображает все настройки для создаваемого расширения, которые вы выбрали в ходе работы Мастера (рис. 6.6). Нажимаем кнопку **Finish**.

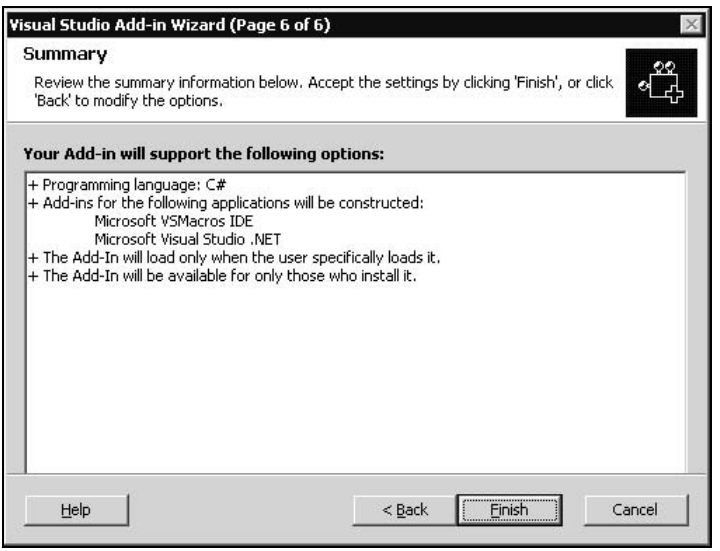


Рис. 6.6. Последний шаг Мастера расширений

Теперь у вас есть полностью функциональная заготовка (выделенная в отдельное решение) для расширения (листинг 6.1).

Листинг 6.1. Заготовка расширения

```
namespace MyAddin3
{
   using System;
   using Microsoft.Office.Core;
   using Extensibility;
   using System.Runtime.InteropServices;
   using EnvDTE;

#region Read me for Add-in installation and setup information.
   //  при запуске, Мастер Add-in подготовит реестр для расширения
   //  в дальнейшем, расширение будет недоступным по таким причинам, как:
   //  а) вы переместили данный проект на другой компьютер
   //  б) вы выбрали 'Yes', когда появилось сообщение об удалении
   //     расширения
   //  в) реестр был поврежден
```

```
    //  вам необходимо заново зарегистрировать расширение, открыв проект
    //  MyAddin21Setup и щелкнув правой кнопкой мыши в окне Solution
    //  Explorer, или выбрав в выпадающем меню пункт Install
#endregion

    ///  <summary>
    ///  объект, реализующий расширения
    ///  </summary>
    ///  <seealso class='IDTExtensibility2' />
    [GuidAttribute("E1B1B93A-4F91-42F1-BBD1-8F1B13865ED8"),
    ProgId("MyAddin3.Connect")]
    public class Connect : Object, Extensibility.IDTExtensibility2
    {
    ///  <summary>
    ///  реализация конструктора объекта расширения
    ///  разместите код инициализации внутри данного метода
    ///  </summary>
    public Connect()
    {
    }
    ///  <summary>
    ///  реализация метода OnConnection интерфейса
    ///  IDTExtensibility2
    ///  получение подтверждения о загрузке расширения
    ///  </summary>
    ///  <param term='application'>
    ///  главный объект host-приложения
    ///  </param>
    ///  <param term='connectMode'>
    ///  описание, как расширение загружается
    ///  </param>
    ///  <param term='addInInst'>
    ///  объект, представляющий данное расширение
    ///  </param>
    ///  <seealso class='IDTExtensibility2' />
    public void OnConnection(object application,
    Extensibility.ext_ConnectMode connectMode,
    object addInInst, ref System.Array custom)
    {
    applicationObject = (_DTE)application;
    addInInstance = (AddIn)addInInst;
    }
```

```
///  <summary>
///  реализация метода OnDisconnection для интерфейса
///  IDTExtensibility2
///  получение подтверждения о выгрузке расширения
///  </summary>
///  <param term='disconnectMode'>
///  описание, как расширение загружается
///  </param>
///  <param term='custom'>
///  массив параметров для host-приложения
///  </param>
///  <seealso class='IDTExtensibility2' />
public void OnDisconnection(Extensibility.ext_DisconnectMode
disconnectMode,
ref System.Array custom)
{
}
///  <summary>
///  реализация метода OnAddInsUpdate интерфейса
///  IDTExtensibility2
///  получение подтверждения о том, что коллекция расширений
///  была изменена
///  </summary>
///  <param term='custom'>
///  массив параметров для host-приложения
///  </param>
///  <seealso class='IDTExtensibility2' />
public void OnAddInsUpdate(ref System.Array custom)
{
}
///  <summary>
///  реализация метода OnStartupComplete интерфейса
///  IDTExtensibility2
///  получение подтверждения о том, что host-приложение
///  завершило загрузку
///  </summary>
///  <param term='custom'>
///  массив параметров для host-приложения
///  </param>
///  <seealso class='IDTExtensibility2' />
public void OnStartupComplete(ref System.Array custom)
{
}
```

```
    ///  <summary>
    ///  реализация метода OnBeginShutdown интерфейса
    ///  IDTExtensibility2
    ///  получение подтверждения о том, что host-приложение
    ///  начало выгружаться
    ///  </summary>
    ///  <param term='custom'>
    ///  массив параметров host-приложения
    ///  </param>
    ///  <seealso class='IDTExtensibility2' />
    public void OnBeginShutdown(ref System.Array custom)
    {
    }
    private _DTE applicationObject;
    private AddIn addInInstance;
    }
}
```

Чтобы расширение позволило выполнять какие-либо функции, вам необходимо написать соответствующий код. Я код писать не буду, т. к. пока вы не изучили *часть III* книги, в которой речь пойдет об языковых конструкциях и особенностях языка C#.

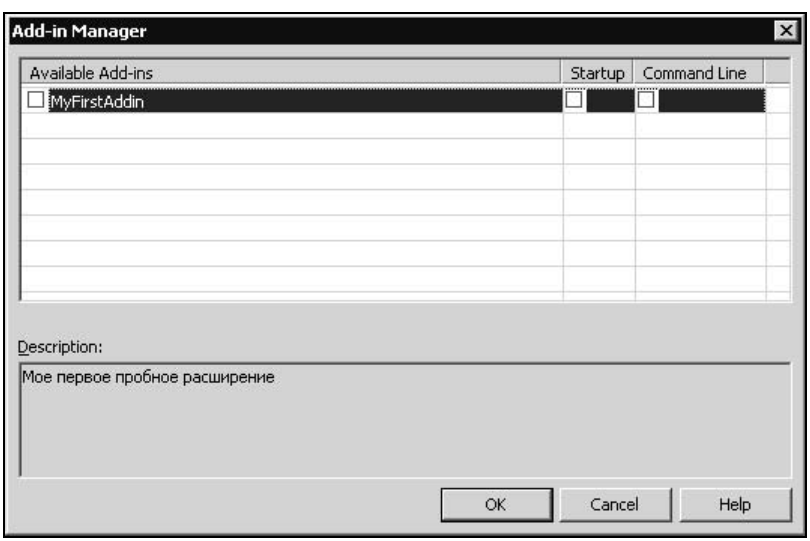


Рис. 6.7. Окно **Add-in Manager**

Для того чтобы посмотреть, какие расширения уже были созданы в среде Visual Studio .NET, используется специальное окно **Add-in Manager**

(рис. 6.7), которое всегда можно вызвать с помощью пункта **Tools | Add-in Manager** главного меню среды разработки.

Кроме создания расширений, вы можете также создавать свои собственные Мастера (Wizard), которые затем можно использовать при создании приложений.

Для создания Мастера необходимо выполнить следующие действия:

1. Создать новый dll-проект.
2. Указать в модуле класса поддержку интерфейса `IDTWizard` таким образом: `Implements IDTWizard`.
3. Добавить необходимый код в процедуре `Execute`.
4. Откомпилировать проект. При этом будет создан dll-файл.
5. Создать специальный vsz-файл, необходимый для того, чтобы приложение активизировалось как Мастер.

Примечание

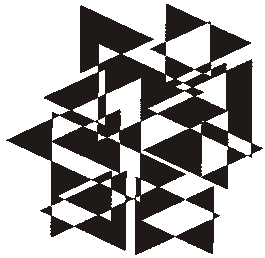
Файл с расширением vsz представляет собой простой текстовый файл, который состоит из трех секций: первая содержит информацию об используемой версии среды Visual Studio, вторая — идентификатор программы Мастера, а третья может содержать необязательные параметры.

Пример типичного содержания vsz-файла (взят из MSDN):

```
VSWizard 7.0
Wizard=VIDWizard.CBlankSiteWizard (or a GUID in the format
                                   {<8>-<4>-<4>-<4>-<12>))
Param=<item1>
Param=<item2>
```

Итак, в данной главе я кратко рассмотрел возможности, предоставляемые средой разработки Visual Studio .NET 2003, для создания расширений самой среды разработки.

В *гл. 7* будут изложены основные принципы работы с проектами и решениями.



ГЛАВА 7

Проекты и решения

В данной главе рассматриваются решения и их составляющие. Вы научитесь добавлять и удалять файлы из решений.

Оперирование проектами и решениями

Среда Visual Studio .NET 2003 поддерживает специальные виды контейнеров, которые называются *решениями* (Solution) и *проектами* (Project).

При создании нового проекта среда Visual Studio .NET автоматически создает новое решение.

Решение — это контейнер, который содержит один или несколько проектов, а также другие файлы, которые необходимы для работы проектов.

Примечание

Решение может и не содержать ни одного проекта, а включать набор различных файлов.

Проект — это набор файлов, которые обычно при компиляции собираются в единый файл с расширением exe или dll. Таким образом, проект — это тоже своеобразный контейнер.

В состав проекта могут входить файлы, ссылки на библиотеки, папки и многое другое.

Состав решения и входящих в него проектов отображается в окне просмотра решения **Solution Explorer** (рис. 7.1).

Это окно (если оно не отображается) можно активировать с помощью комбинации клавиш <Ctrl>+<Alt>+<L> или пункта **View | Solution Explorer** главного меню среды разработки.

Рис. 7.1. Окно **Solution Explorer**

Создание решений и проектов

Для создания нового решения, не содержащего в себе ни одного проекта или файла, выберите пункт **File | New Blank Solution** в главном меню среды разработки. Затем в открывшемся диалоговом окне **New Project** (рис. 7.2) введите имя нового решения в поле **Name** и укажите его расположение на диске в поле **Location**.

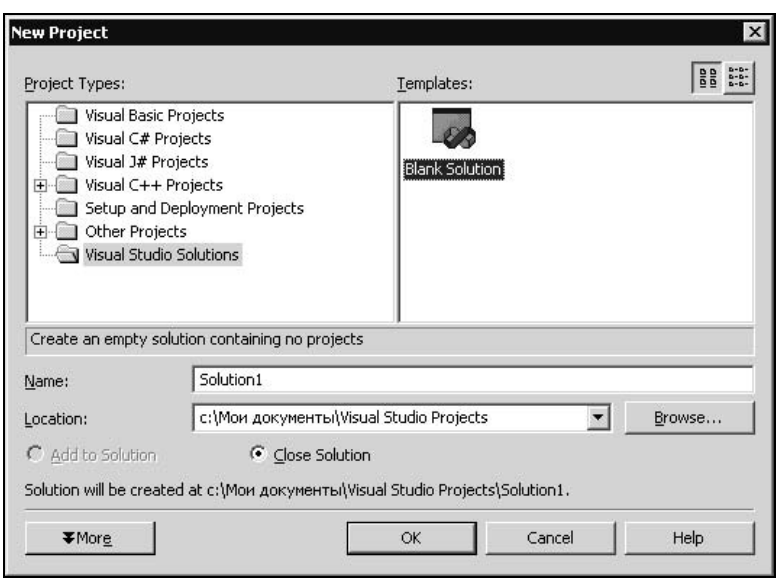


Рис. 7.2. Создание нового решения

Создание нового проекта осуществляется аналогично, с помощью пункта **File | New Project** главного меню Visual Studio .NET 2003.

Добавление файлов и элементов в решения и проекты

Вы можете добавить в готовое решение новые файлы или проекты. Для этого в окне **Solution Explorer** (см. рис. 7.1) щелкните правой кнопкой мыши на имени решения и в ниспадающем меню выберите пункт **Add**. Откроется подменю, в котором можно выбрать один из следующих пунктов:

- **New Project** — добавление нового проекта;
- **Existing Project** — добавление уже существующего проекта;
- **Existing Project From Web** — добавление уже существующего проекта из Интернета;
- **Add New Item** — добавление нового элемента (текстового, HTML, изображения, XML, курсора, пиктограммы и т. д);
- **Add Existing Item** — добавление уже существующего элемента.

Аналогично производится добавление элементов в проект. Для этого щелкните правой кнопкой мыши на названии проекта, в окне **Solution Explorer** и в ниспадающем меню выберите пункт **Add** (рис. 7.3). Откроется подменю, в котором можно выбрать один из следующих пунктов:

- **Add New Item** — добавление нового элемента (формы, файла на C++, HTML-файла, заголовка и т. д.);
- **Add Existing Item** — добавление уже существующего элемента;
- **New Folder** — создание новой папки внутри проекта;
- **Add Class** — добавление класса из внешней библиотеки;
- **Add Resource** — добавление ресурса.

Для удаления элемента из проекта или решения щелкните на элементе правой кнопкой мыши и в появившемся меню выберите пункт **Remove**.

Проекты и решения хранятся в папках на диске по указанному при их создании пути. На рис. 7.4 показано решение **MyNew1**, содержащее один проект с именем MyNew1 в окне **Solution Explorer**, и оно же, открытое с помощью проводника Windows.

Такая организация очень рациональна. Все файлы, относящиеся к одному проекту или решению, хранятся в одной папке, что облегчает работу с ними.

Мы еще вернемся к обсуждению проектов и решений в дальнейшем, при создании приложений.

В *гл. 8* будут рассмотрены особенности встроенного в среду Visual Studio .NET 2003 редактора кода.

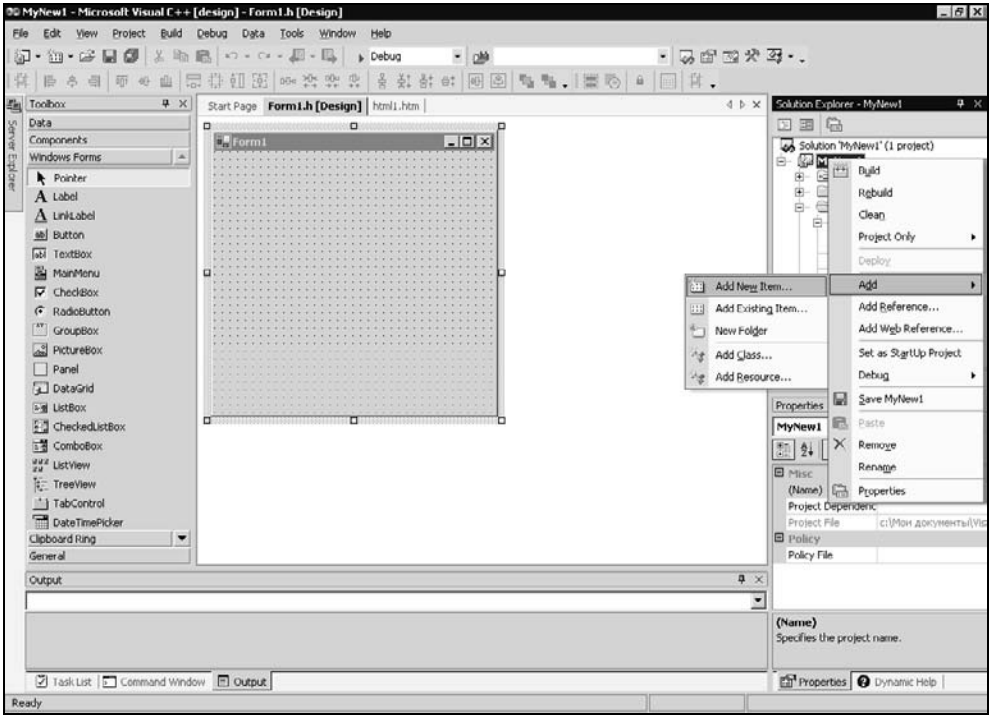


Рис. 7.3. Добавление нового элемента в проект

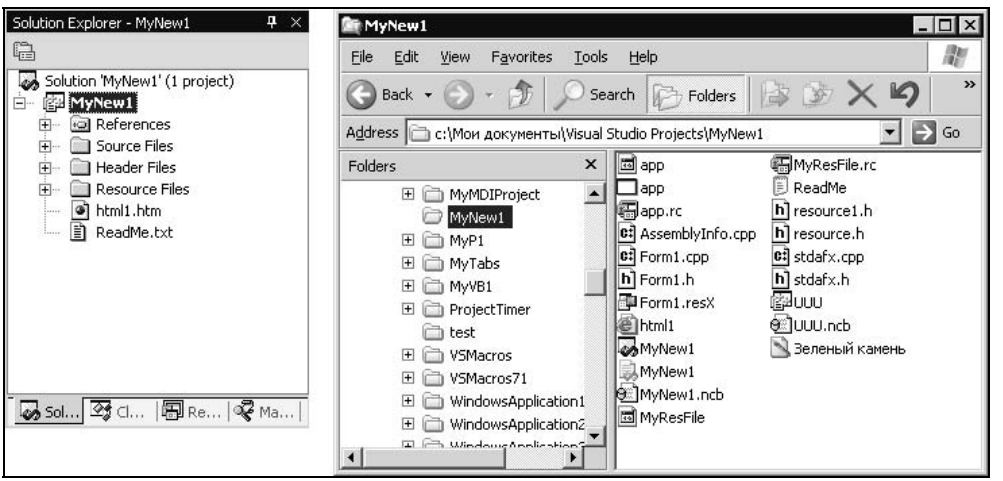
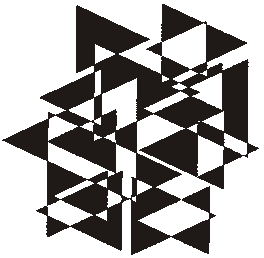


Рис. 7.4. Решение **MyNew1** в окне **Solution Explorer** и проводнике Windows

ГЛАВА 8



Работа с редактором кода

В данной главе мы рассмотрим возможности, которые предоставляет редактор кода Visual Studio .NET 2003. Изучим механизм IntelliSense, а также панели инструментов.

Редактор кода Visual Studio .NET

Редактор кода Visual Studio .NET осуществляет множество специфических функций, которые облегчают работу с кодом приложений. Рассмотрим эти функции.

Редактор кода позволяет сворачивать фрагменты программы и отображать на их месте только заголовки. Значки +, находящиеся слева от некоторых строк, свидетельствуют о том, что данный фрагмент программы свернут. На рис. 8.1 показан свернутый фрагмент программы с заголовком `Windows Form Designer generated code`.

Для его "разворачивания" достаточно просто щелкнуть левой кнопкой мыши на значке +. При этом откроется *область* (Region), начинающаяся с ключевого слова `#Region "Имя_региона"` и заканчивающаяся ключевым словом `#End Region` (рис. 8.2).

Примечание

Вы можете самостоятельно выделять фрагменты своих программ в регионы с помощью ключевых слов, а затем сворачивать их. Внутри областей могут также находиться области; количество вложений областей друг в друга не ограничено.

Окно редактора кода позволяет использовать многоэлементный буфер обмена (как в Microsoft Office 2000 или XP), позволяющий хранить сразу несколько скопированных "кусков" текста. Буфер обмена позволяет сохранять

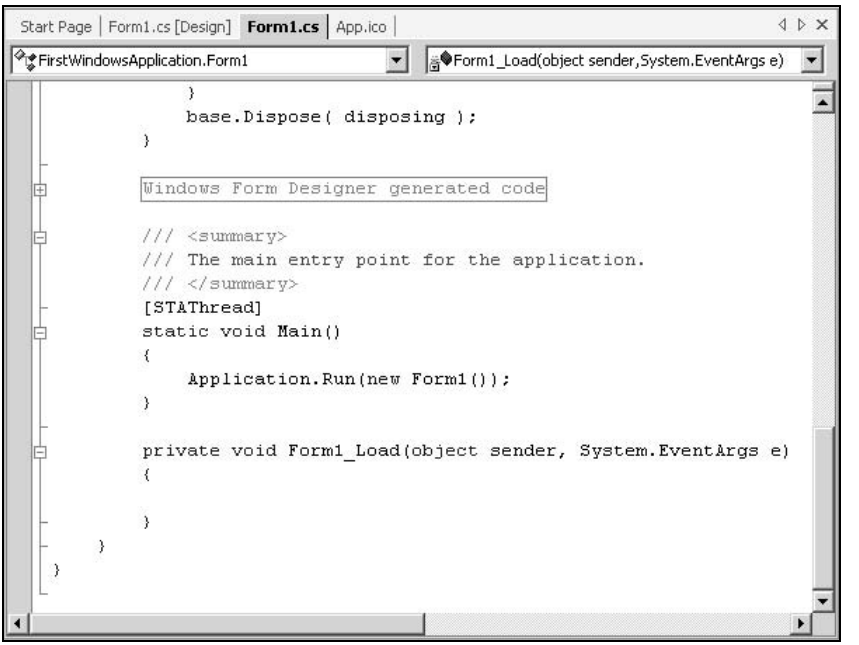


Рис. 8.1. Свернутый фрагмент программы

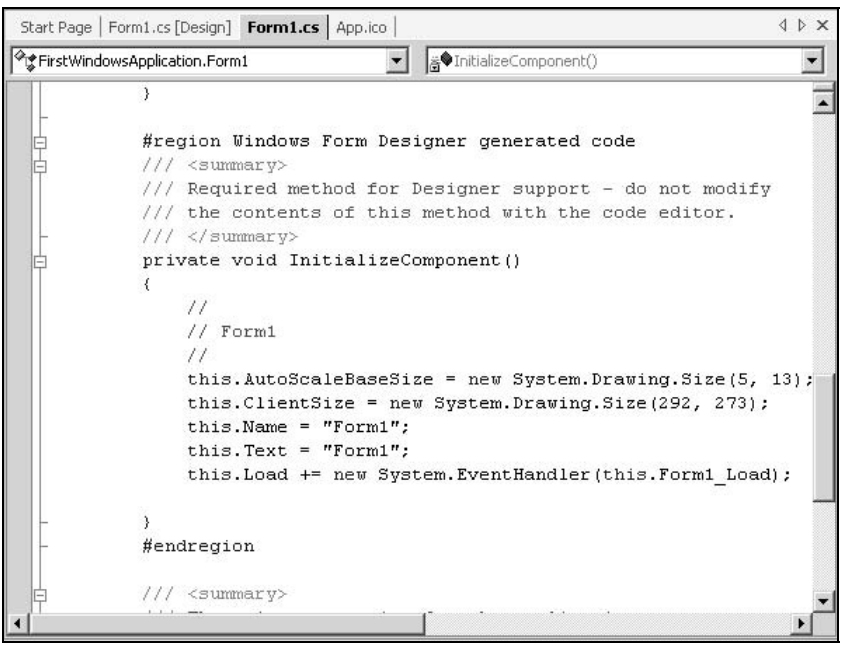


Рис. 8.2. Развернутая область (Region)

до 15 текстовых элементов. Запоминать текст можно стандартными комбинациями клавиш: <Ctrl>+<C> — для копирования и <Ctrl>+<X> — для вырезания текста. Вставка последнего из скопированных текстового элемента из буфера обмена производится с помощью комбинации клавиш <Ctrl>+ +<V>. Для последовательного перебора содержимого буфера обмена используйте несколько раз комбинацию клавиш <Ctrl>+<Shift>+<V>.

Кроме того, для временного хранения произвольного участка текста, с целью последующей его вставки в другое место программы, вы можете использовать панель элементов (расположена слева от окна редактора кода). Для этого достаточно просто выделить нужный участок текста и перетащить его с помощью мыши на панель элементов (рис. 8.3). Для вставки скопированного на панель элементов текста в нужное место программы достаточно просто перетащить его в это место из панели элементов. Удалить ненужный текстовый участок из панели элементов можно, щелкнув по нему правой кнопкой мыши и выбрав в раскрывающемся меню пункт **Delete**.

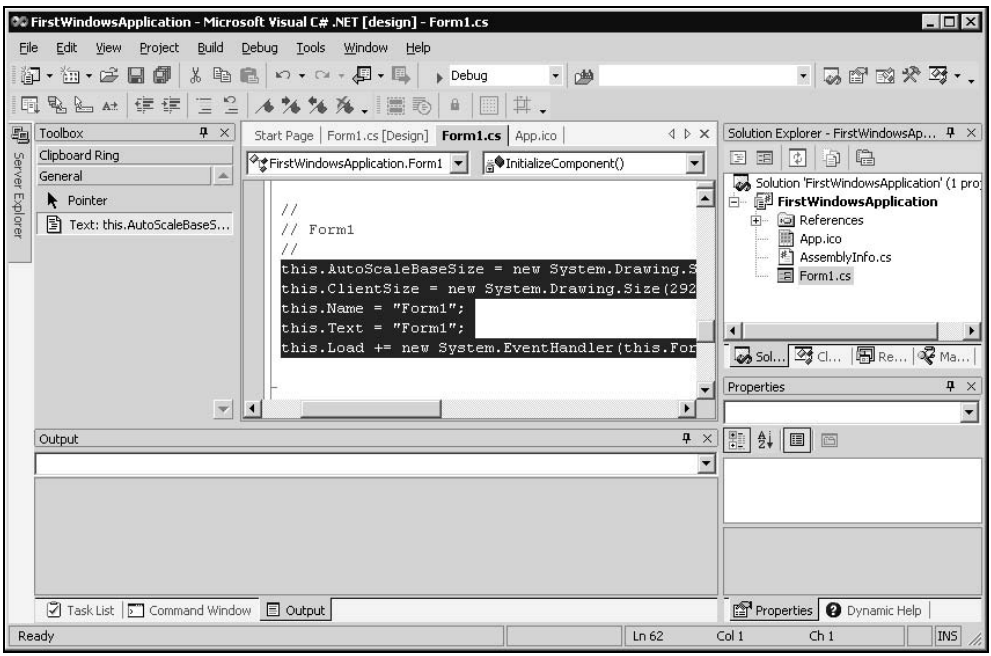


Рис. 8.3. Фрагмент кода на панели элементов

Использование механизма IntelliSense

Механизм *IntelliSense* предоставляет программисту обширные дополнительные возможности редактирования кода программ.

Перечислим эти возможности:

- отображение списка членов *(List Members)* для класса, структуры, пространства имен и других элементов кода;
- отображение информации о параметрах *(Parameter Info)* для методов и функций;
- отображение краткого описания *(Quick Info)* элемента кода программы;
- завершение слов *(Complete Word)* для команд и имен функций;
- автоматическое сопоставление скобок *(Automatic Brace Matching)*.

Рассмотрим эти возможности более подробно.

Отображение списка членов включается автоматически, когда вы указали имя класса или структуры и выбрали один из следующих операторов:

- . (точка) для экземпляра класса;
- -> (указатель) для указывания на экземпляр класса;
- :: для класса, структуры или пространства имен.

При этом механизм IntelliSense отобразит доступные члены в списке с линейкой прокрутки (рис. 8.4).

Вы можете вставить нужный член класса с использованием клавиш <Tab>, <Enter>, комбинации клавиш <Ctrl>+<Enter> или с помощью двойного щелчка на нужном члене класса. Для закрытия списка используйте клавишу <Esc>.

Для того чтобы принудительно отобразить список членов применяйте используйте комбинацию клавиш <Ctrl>+<J> или пункт **Edit | IntelliSense | List Members** главного меню среды разработки.

Отображение информации о параметрах для методов и функций также включается автоматически. После того, как вы наберете имя функции, имеющей параметры, на экране появится информация об этих параметрах (рис. 8.5).

Вы можете принудительно отобразить информацию о параметрах с помощью комбинации клавиш <Ctrl>+<Shift>+<Пробел> или пункта главного меню среды разработки **Edit | IntelliSense | Parameter Info**.

Отображение краткого описания происходит при наведении указателя мыши на нужный элемент кода программы (рис. 8.6).

Механизм *завершения слов* активизируется с помощью комбинации клавиш <Ctrl>+<Пробел> после того, как вы наберете несколько букв из имени функции. При этом появится список доступных членов класса и будет выбран тот член класса, который начинается на набранные вами буквы (рис. 8.7).

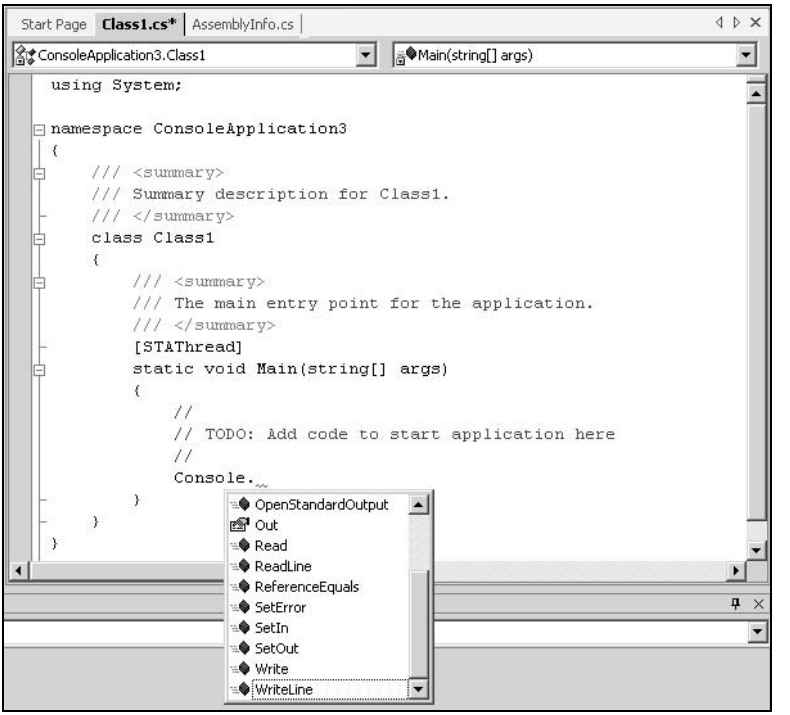


Рис. 8.4. Отображение списка членов класса `Console`

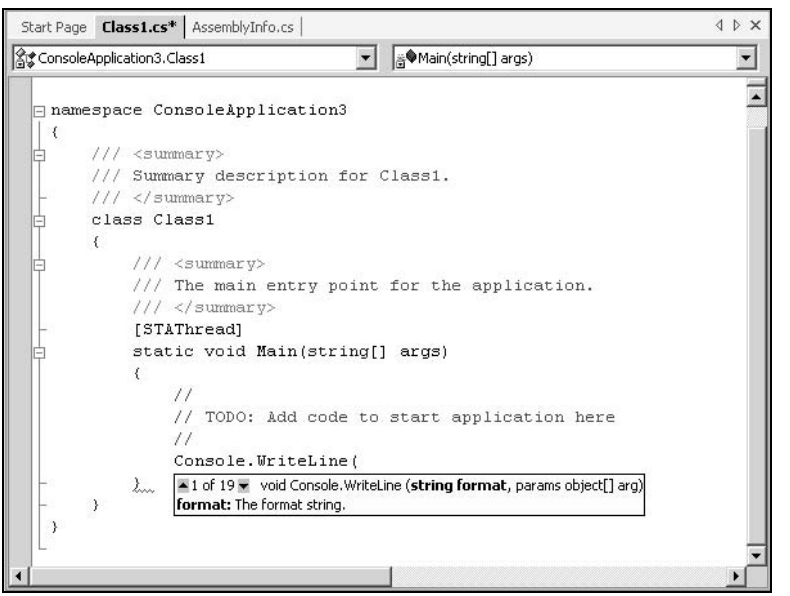


Рис. 8.5. Отображение информации о параметрах функции `WriteLine` класса `Console`

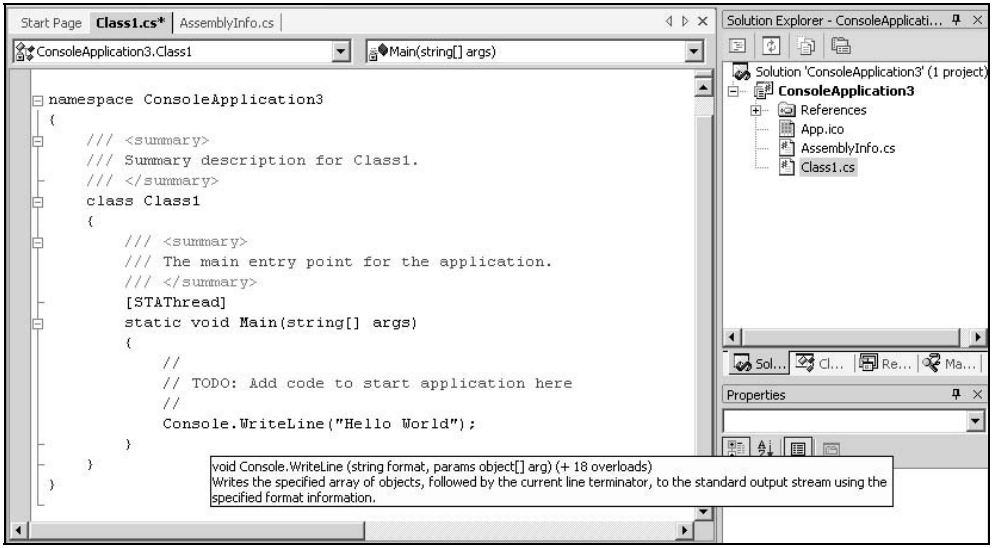


Рис. 8.6. Краткое описание функции `WriteLine` класса `Console`

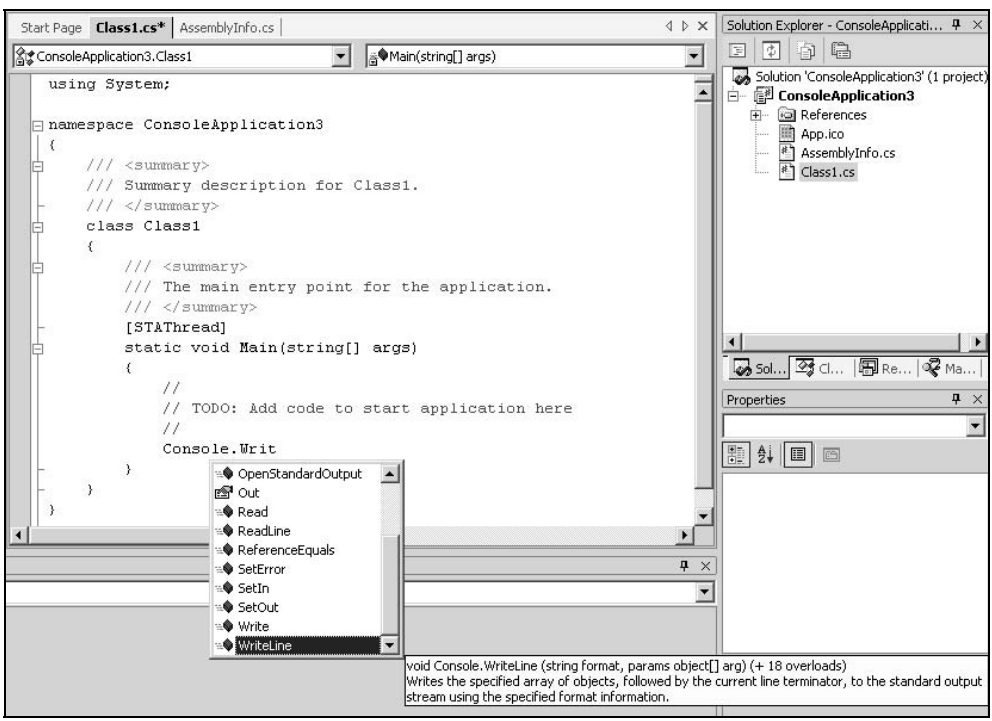


Рис. 8.7. Завершение слова `WriteLine`

Как вы можете видеть на рис. 8.7, кроме списка членов класса отображается еще и краткая информация о данной функции.

Ну и последняя возможность механизма IntelliSense — автоматическое сопоставление скобок, проявляется следующим образом. Когда вы набираете закрывающую скобку (вида `)`, `]`, `}` или `#endif`), она будет выделяться полужирным начертанием вместе со своей открывающей скобкой. Например, так, как показано на рис. 8.8.

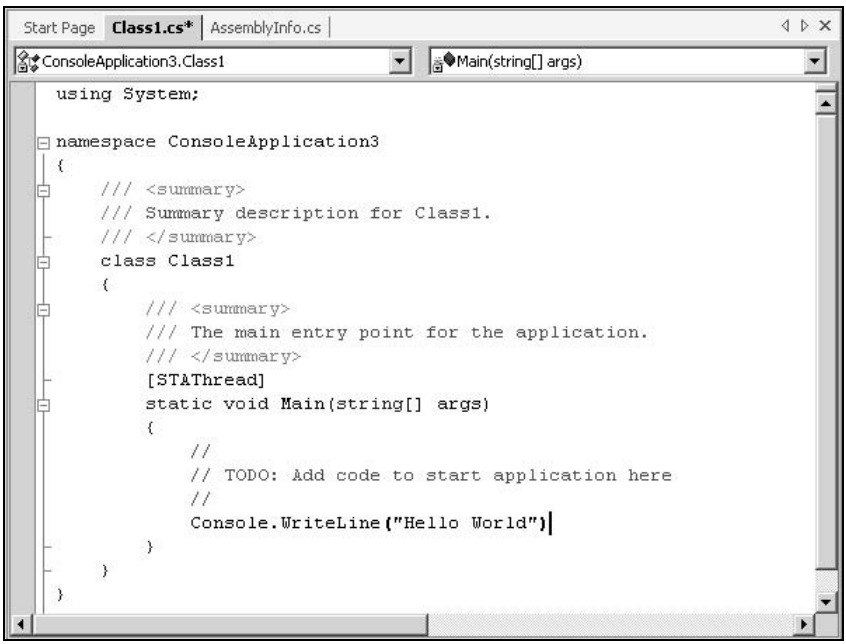


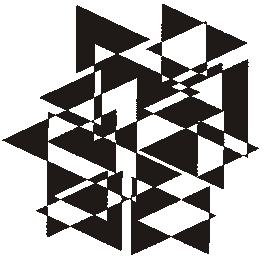
Рис. 8.8. Автоматическое сопоставление скобок

Механизм IntelliSense может быть недоступен в некоторых случаях, а именно:

- если в тексте программы перед вводимым кодом имеются ошибки;
- если вы пишете комментарий;
- если вы пишете строковое значение, заключенное в кавычки;
- если IntelliSense был отключен с помощью окна настроек (появляется при выполнении пункта **Tools | Options** главного меню Visual Studio .NET 2003).

В этой главе мы кратко рассмотрели особые возможности редактора кода среды Microsoft Visual Studio .NET 2003. В *гл. 9* будет описана работа с файлами ресурсов.

ГЛАВА 9



Работа с файлами ресурсов

В этой главе рассмотрим, какие файлы ресурсов поддерживаются средой Visual Studio .NET 2003. Кроме того, здесь же приводится краткое описание основных редакторов ресурсов, входящих в состав среды разработки.

Файлы ресурсов

В ваши проекты, скорее всего, будут входить не только файлы с расширениями exe и dll, но и другие файлы с дополнительной информацией, которые относятся к так называемым *ресурсным файлам*.

Примечание

Файлы с расширениями exe и dll также являются ресурсными.

В табл. 9.1 перечислены типы файлов ресурсов, которые вы можете создавать и редактировать с помощью среды разработки Visual Studio .NET 2003.

Таблица 9.1. *Типы файлов ресурсов, поддерживаемые средой разработки*

Расширение файла ресурсов	Краткое описание
rc	Файл сценария (script)
rct	Файл шаблона (template)
res	Файл ресурсов
exe	Исполняемый файл. Редактируется только в операционных системах, построенных на платформе Windows NT. В операционных системах Windows 9*x*/ME может быть только открыт, но не изменен

Таблица 9.1 *(окончание)*

Расширение файла ресурсов	Краткое описание
dll	Файл динамически присоединяемой библиотеки. По возможностям редактирования аналогичен файлу с расширением exe
bmp	Графический файл обычного для Windows формата
ico	Графический файл, предназначенный для хранения изображения пиктограммы
dib	Файл ресурсов, хранящий сведения о панели инструментов
cur	Файл, хранящий образ указателя мыши (курсора)

Создание файлов ресурсов

Для создания нового файла ресурсов используйте пункт **Project | Add New Item** главного меню среды разработки или комбинацию клавиш <Ctrl>+ +<Shift>+<A>. Появится диалоговое окно добавления нового файла в проект (рис. 9.1).

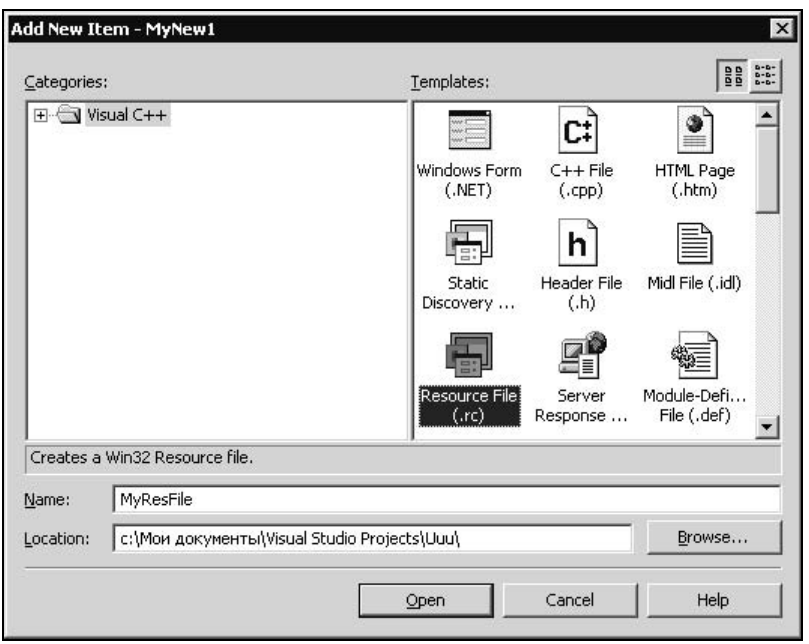


Рис. 9.1. Добавление нового файла в проект

Выберите пиктограмму **Resource File** в поле **Templates**, как показано на рис. 9.1, и в поле **Name** впишите имя создаваемого файла ресурсов. Затем нажмите кнопку **Open**. Теперь вы можете создавать и добавлять новые ресурсы в созданный файл с расширением rc.

Примечание

Вы можете создавать новые файлы ресурсов с расширением rc только внутри какого-либо проекта. Создавать такие файлы отдельно от проекта нельзя (среда Visual Studio .NET 2003 не предоставляет программистам такой возможности).

Для добавления ресурсов в файл с расширением rc, сначала выберите его в окне **Resource View** (рис. 9.2). Это окно активизируется с помощью пункта **View | Resource View** главного меню среды разработки.

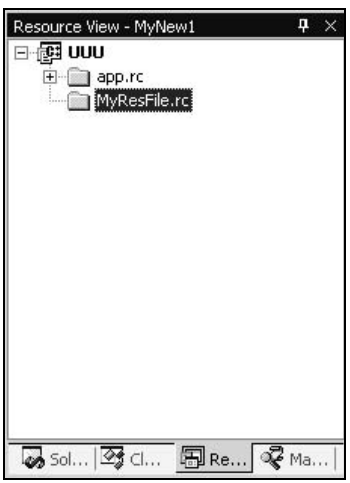


Рис. 9.2. Окно **Resource View**

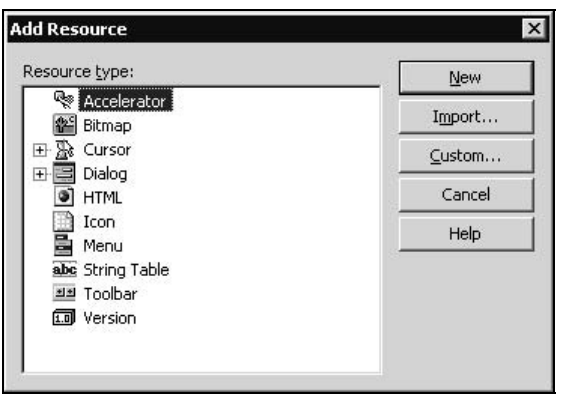


Рис. 9.3. Диалоговое окно **Add Resource**

Затем выберите пункт **Edit | Add Resource** в главном меню среды разработки или щелкните правой кнопкой мыши на имени файла с расширением rc. В ниспадающем меню выберите пункт **Add Resource**. Появится одноименное диалоговое окно выбора ресурса (рис. 9.3).

Таким образом, во вновь созданный файл с расширением rc вы можете добавить один из типов ресурсов, приведенных в поле **Resource type** (см. рис. 9.3). Рассмотрим ресурсы данных типов:

- **Accelerator** — установка комбинации "горячих" клавиш;
- **Bitmap** — графическое изображение;
- **Cursor** — указатель мыши;

- **Dialog** — стандартное диалоговое окно Windows;
- **HTML** — текст в формате HTML;
- **Icon** — графическое изображение пиктограммы;
- **Menu** — меню;
- **String Table** — таблица, для хранения текстовой информации;
- **Toolbar** — панель инструментов;
- **Version** — информация о версии проекта.

Вы можете просматривать некоторые типы ресурсов без их непосредственного открытия (в режиме предварительного просмотра **Preview**). Для этого укажите нужный ресурс в файле с расширением rc при помощи окна **Resource View**, а затем выберите пункт **View | Property Pages** главного меню или используйте комбинацию клавиш <Shift>+<F4>. Например, для просмотра графического изображения (рис. 9.4).

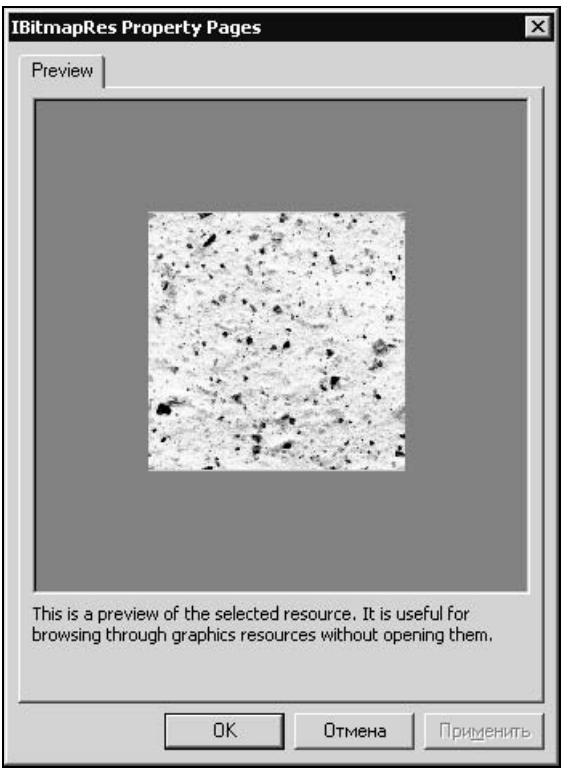


Рис. 9.4. Предварительный просмотр графического ресурса

Редакторы ресурсов

В состав Visual Studio .NET 2003 входят девять специальных редакторов ресурсов:

- *Accelerator Editor* (редактор "горячих" клавиш) — может использоваться при создании проектов на языке Visual C++;
- *Binary Editor* (редактор двоичных файлов) — применяется для редактирования двоичных данных в проектах на языках Visual C++, Visual C# и Visual Basic;
- *Dialog Editor* (редактор диалоговых окон) — используется при создании проектов на языке Visual C++;
- *HTML Editor* (редактор HTML-страниц) — позволяет редактировать HTML-страницы;
- *Image Editor* (редактор рисунков) — используется для просмотра и редактирования графических изображений, пиктограмм, курсоров и других графических файлов;
- *Menu Editor* (редактор меню) — применяется для редактирования меню в проектах на языке Visual C++;
- *String Editor* (редактор таблиц, содержащих текстовую информацию) — используется при создании проектов на языке Visual C++;
- *Toolbar Editor* (редактор панелей инструментов) — позволяет редактировать панели инструментов для проектов на языке Visual C++. Является частью редактора Image Editor;
- *Version Information Editor* (редактор информации о версии) — используется при создании проектов на языке Visual C++.

Редактор Accelerator Editor

Для начала, напишу несколько слов о *таблицах "горячих" клавиш*. Такие таблицы содержат список "горячих" клавиш и идентификаторов команд, связанных с данными "горячими" клавишами.

Обычно "горячие" клавиши применяются для быстрого доступа к пунктам меню программы или к пиктограммам панелей инструментов.

С помощью редактора "горячих" клавиш вы можете устанавливать их свойства, связывать "горячие" клавиши с пунктами меню приложений, редактировать таблицы "горячих" клавиш, и многое другое.

На рис. 9.5 показана таблица "горячих" клавиш, открытая в редакторе Accelerator Editor.

Вы можете устанавливать свойства "горячих" клавиш, выбрав их в окне редактора и используя окно свойств **Properties**. Эти свойства и их описание приведены в табл. 9.2.

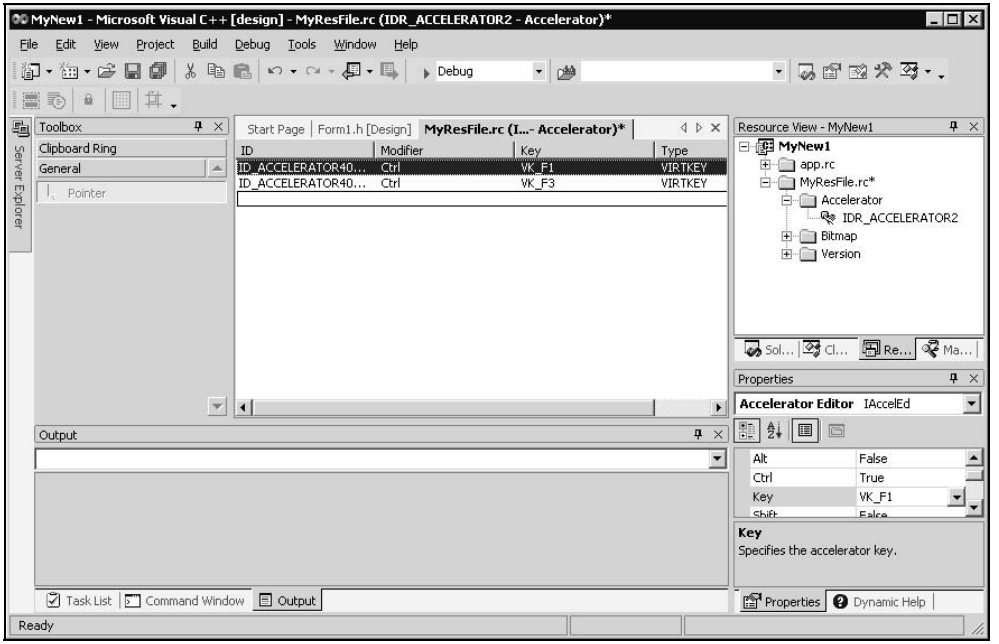


Рис. 9.5. Таблица "горячих" клавиш в окне редактора Accelerator Editor

Таблица 9.2. *Свойства "горячих" клавиш*

Свойство	Краткое описание
`Alt`	Если установлено значение `True`, то для активизации "горячей" клавиши при выполнении приложения необходимо нажать клавишу <Alt> и клавишу, указанную в свойстве `Key`. Если другие свойства (`Ctrl` и/или `Shift`) имеют значения `True`, то их также необходимо нажимать вместе с клавишей <Alt> и клавишей, указанной в свойстве `Key`
`Ctrl`	Если установлено значение `True`, то для активизации "горячей" клавиши при выполнении приложения необходимо нажать клавишу <Ctrl> и клавишу, указанную в свойстве `Key`. Если другие свойства (`Alt` и/или `Shift`) имеют значения `True`, то их также нужно нажать вместе с <Ctrl> и клавишей, указанной в свойстве `Key`
`ID`	Содержит значение, которое будет передаваться программе, когда пользователь нажмет комбинацию "горячих" клавиш. Для того, чтобы при нажатии "горячих" клавиш активизировался соответствующий пункт меню программы, укажите такое же значение в свойстве `ID` для данного пункта меню
`Key`	Позволяет указать клавишу, которая будет использоваться как "горячая" клавиша (возможно, совместно с другими клавишами, такими как <Alt>, <Ctrl> или <Shift>)

Таблица 9.2 *(окончание)*

Свойство	Краткое описание
`Shift`	Если установлено значение `True`, то для активизации "горячей" клавиши при выполнении приложения необходимо нажать клавишу <Shift> и клавишу, указанную в свойстве `Key`. Если другие свойства (`Ctrl` и/или `Alt`) имеют значения `True`, то их также нужно нажать вместе с клавишей <Shift> и клавишей, указанной в свойстве `Key`
`Type`	Определяет, какие комбинации клавиш будут использоваться: виртуальные клавиши (virtual key) или ASCII-значения клавиш

Примечание

Если ни одно из свойств `Alt`, `Ctrl` и `Shift` не имеет значение `True`, то пользователю для активизации "горячей" клавиши достаточно будет нажать ту, которая указана в свойстве `Key`.

Редактор Binary Editor

Данный редактор позволяет просматривать и редактировать двоичные файлы, а также выполнять поиск внутри таких файлов.

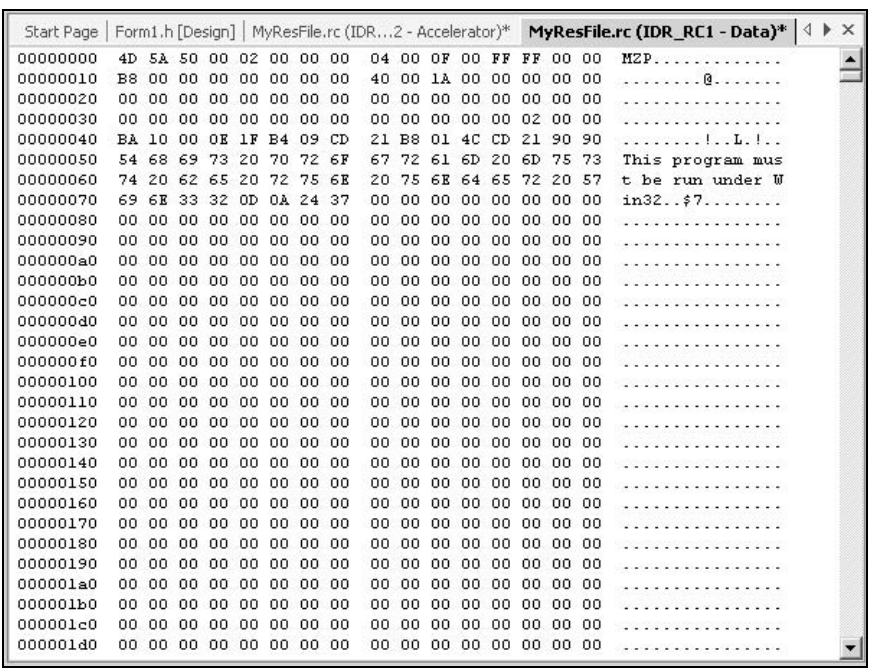


Рис. 9.6. Файл Far.exe, открытый в окне редактора Binary Editor

Редактор Binary Editor представляет информацию в шестнадцатеричной форме, а также в символьном виде.

На рис. 9.6 показан открытый в окне редактора Binary Editor файл Far.exe.

Крайний левый столбец отображает адрес, начиная с нулевого (начало файла). Два следующих широких столбца отображают данные в шестнадцатеричной форме, находящиеся по этим адресам. Крайний правый столбец отображает те же самые данные в символьном виде (ASCII-формат).

Работа с этим редактором не представляет сложности и ничем не отличается от других подобных редакторов.

Редактор Dialog Editor

Как уже было сказано ранее, этот редактор позволяет создавать диалоговые окна, при разработке приложений на языке Visual C++. Вид диалогового окна представлен на рис. 9.7.

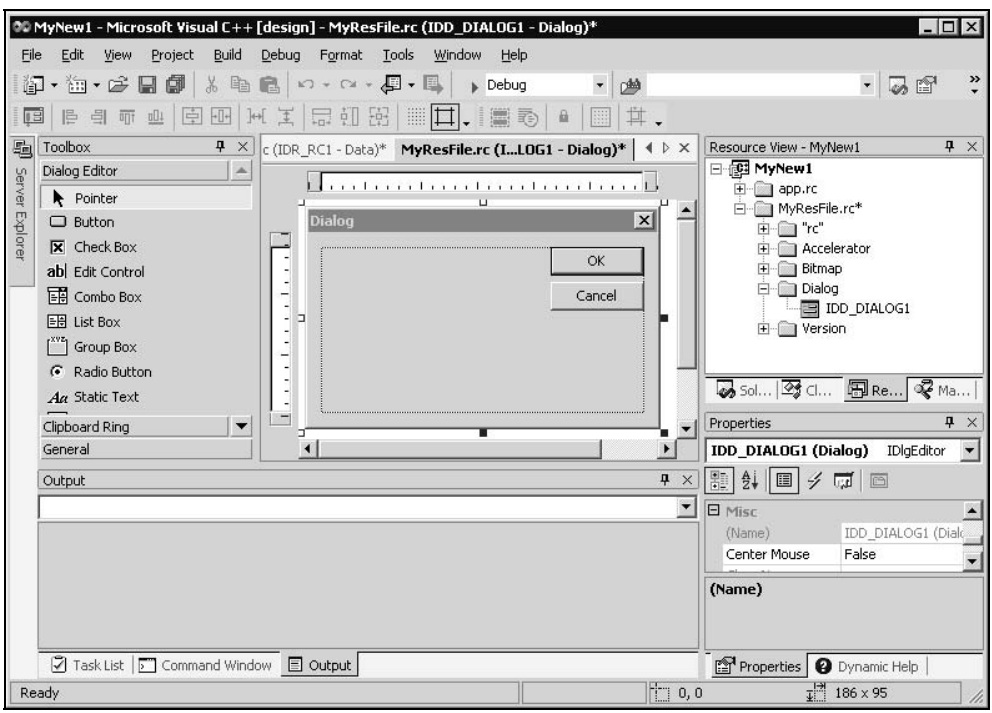


Рис. 9.7. Диалоговое окно в процессе разработки

Работа с редактором Dialog Editor не представляет особой сложности. Все аналогично визуальной разработке простых форм приложений.

Единственное, что можно отметить, — появление над окном редактора панели инструментов **Dialog Editor Toolbar**. Большинство пиктограмм, расположенных на данной панели инструментов, предназначены для выравнивания и расположения элементов диалогового окна. Первая пиктограмма слева на панели инструментов (**Test Dialog**) позволяет просмотреть диалоговое окно в действии.

Редактор HTML Editor

Этот редактор позволяет создавать и редактировать файлы в формате HTML. Редактор поддерживает два режима редактирования:

- *Design* (режим дизайнера) — посредством данного режима вы можете добавлять на HTML-страницу элементы управления визуальным способом с помощью панели HTML слева от окна редактора;
- *HTML* (режим текста) — в этом режиме вы можете просматривать и редактировать текст HTML-страницы.

Для переключения между режимами служат закладки **Design** и **HTML** в нижней части окна редактора HTML Editor.

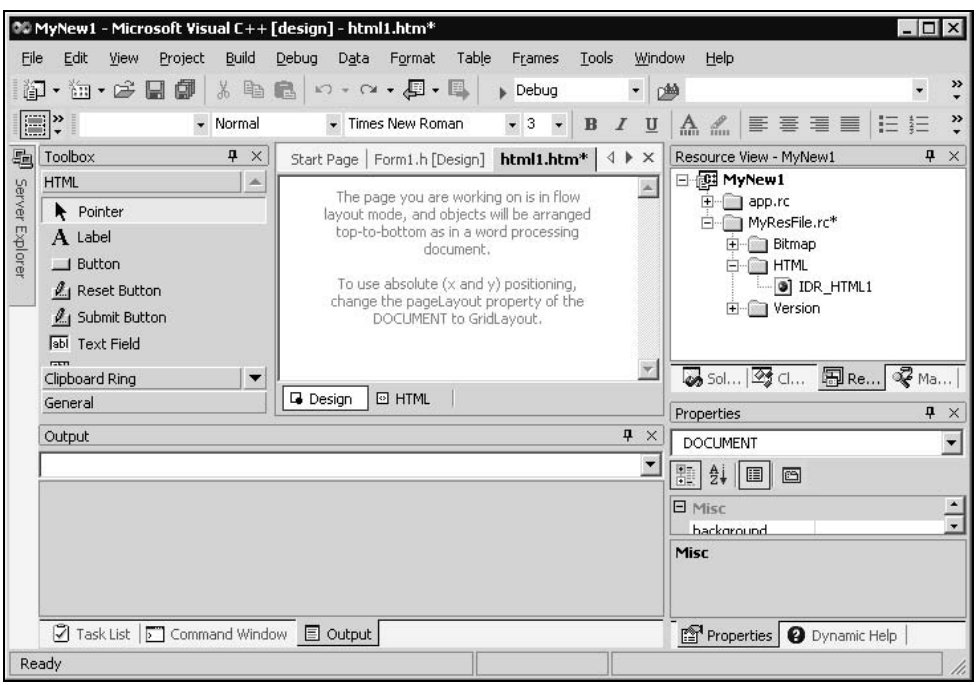


Рис. 9.8. Новая HTML-страница в окне редактора HTML Editor

На рис. 9.8 показана страница редактора HTML Editor в режиме Design (без элементов управления).

Редактор Image Editor

Этот редактор позволяет создавать и редактировать графические файлы различных форматов. Он обладает хорошими возможностями и практически ничем не уступает другим графическим редакторам.

На рис. 9.9 показан файл с расширением bmp, открытый с помощью редактора Image Editor.

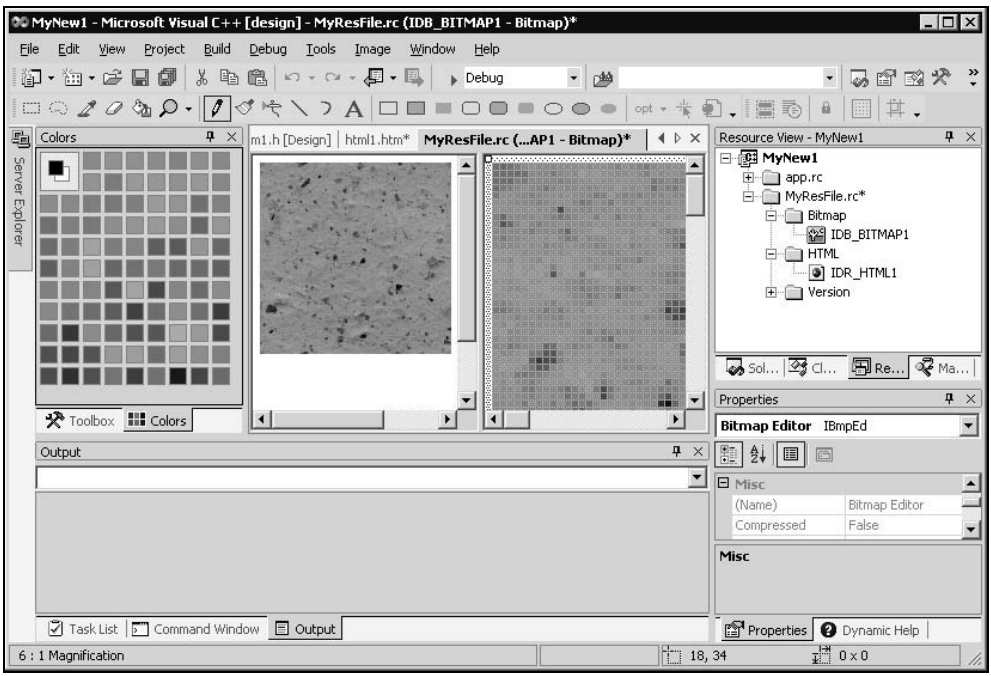


Рис. 9.9. Файл в формате BMP, открытый в редакторе Image Editor

Редактор Menu Editor

Этот редактор позволяет создавать меню для проектов на языке Visual C++. Разрабатываемое меню представляется в редакторе Menu Editor в визуальной форме, как показано на рис. 9.10.

Для каждого пункта меню в окне свойств (**Properties**) вы можете установить необходимые значения свойств. Эти свойства и их краткое описание приведены в табл. 9.3.

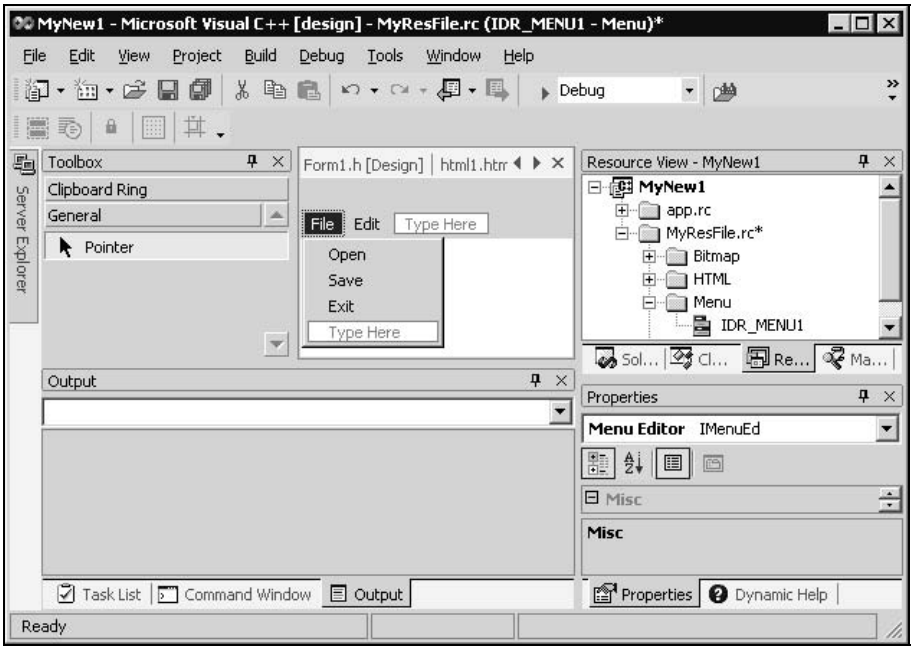


Рис. 9.10. Разработка меню в окне редактора Menu Editor

Таблица 9.3. *Свойства пунктов меню*

Свойство	Краткое описание
`Break`	Может принимать одно из трех значений: `None` — без разделений; `Column` — отделяет предыдущие пункты меню от данного пункта горизонтальной линией; `Bar` — отделяет предыдущие пункты меню от данного пункта вертикальной линией
`Caption`	Содержит текст, который будет отображаться в данном пункте меню
`Checked`	Если установлено значение `True`, то рядом с текстом, отображенным в данном пункте меню, будет стоять галочка. По умолчанию имеет значение `False`
`Enabled`	Если установлено значение `True`, то данный пункт меню будет доступен пользователю, в противном случае этот пункт меню будет закрыт (а текст выделен серым цветом)
`Grayed`	Если установлено значение `True`, то данный пункт меню будет закрыт и полностью заштрихован серыми полосками
`Help`	Определяет, будут ли выравниваться пункты меню по правому краю. По умолчанию имеет значение `False`

Таблица 9.3 *(окончание)*

Свойство	Краткое описание
`ID`	Содержит значение, позволяющее определить действие, выполняемое данным пунктом меню. Для использования "горячих" клавиш — установите необходимое значение, аналогичное значению свойства `ID` для "горячих" клавиш
`Popup`	Определяет, будет ли меню всплывающим (раскрывающимся). По умолчанию — `False`, что означает обычное, не всплывающее меню
`Prompt`	Содержит текст, который будет отображаться при выделении пользователем данного пункта меню
`Right to Left Justify`	Выравнивает пункты меню по правому краю во время выполнения программы. По умолчанию `False` — не выравнивать
`Right to Left Order`	Применяется для отображения текста справа налево, для некоторых арабских и других языков. По умолчанию `False` — отображать текст слева направо
`Separator`	Если установлено значение `True`, то вместо пункта меню будет отображаться разделительная полоса. По умолчанию имеет значение `False`

Редактор String Editor

Данный редактор позволяет просматривать, создавать и редактировать *таблицы строк*. Такие таблицы содержат список идентификаторов (столбец **ID**), числовых значений (столбец **Value**) и текста (столбец **Caption**) для строк приложения. Например, подсказки, отображаемые в строках состояний, создаются и хранятся в строковых таблицах.

Такой подход облегчает создание многоязыковых приложений. Для перевода готового приложения на другой язык вам достаточно перевести строки в таблице строк, без исправлений в исходном коде приложения.

На рис. 9.11 показана таблица, состоящая из двух строк, открытая в окне редактора String Editor.

Вы можете вставлять в текст строки специальные символы и символы форматирования, приведенные в табл. 9.4. Для этого вставьте их в нужном месте текста (столбец **Caption**).

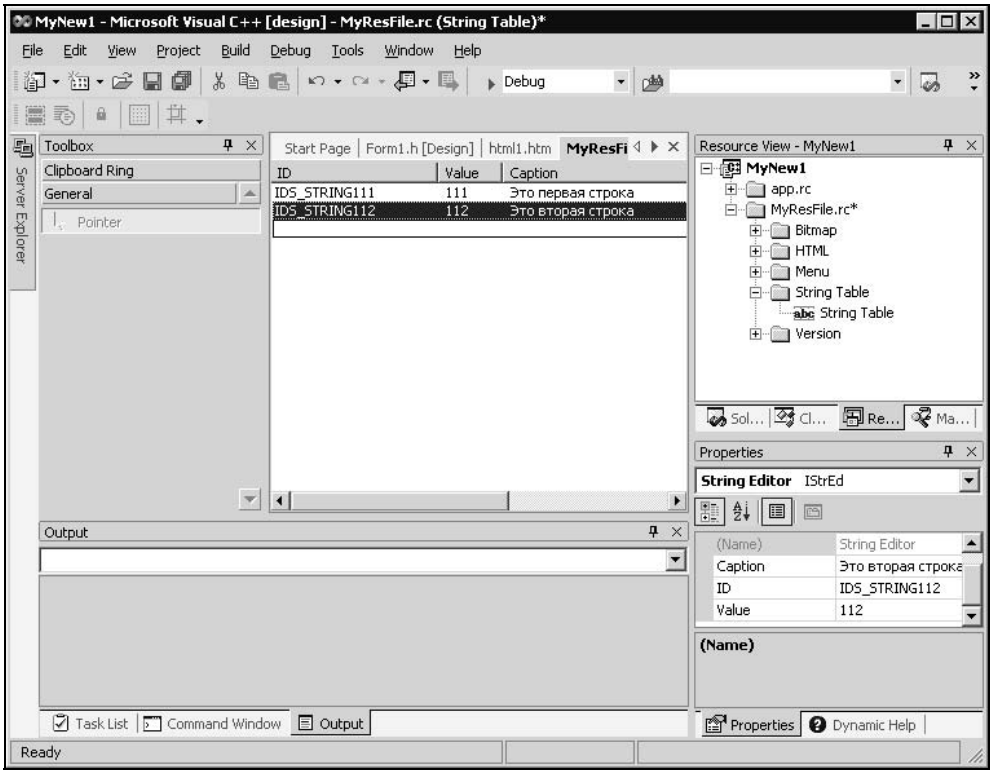


Рис. 9.11. Таблица строк в окне редактора String Editor

Таблица 9.4. *Символы форматирования и специальные символы*

Символы	Результат набора символов
`\n`	Переход на новую строку
`\r`	Возврат каретки
`\t`	Табуляция
`\\`	Обратный слэш (\)
`\ddd`	Символ ASCII, состоящий из трех цифр в восьмеричной системе счисления
`\a`	Звуковой сигнал

Редактор Toolbar Editor

Этот редактор применяется для создания панелей инструментов и их кнопок, конвертирования графических файлов для вставки на панелях инструментов, а также создания подсказок к кнопкам панелей инструментов.

На рис. 9.12 показан вид панели инструментов в процессе создания.

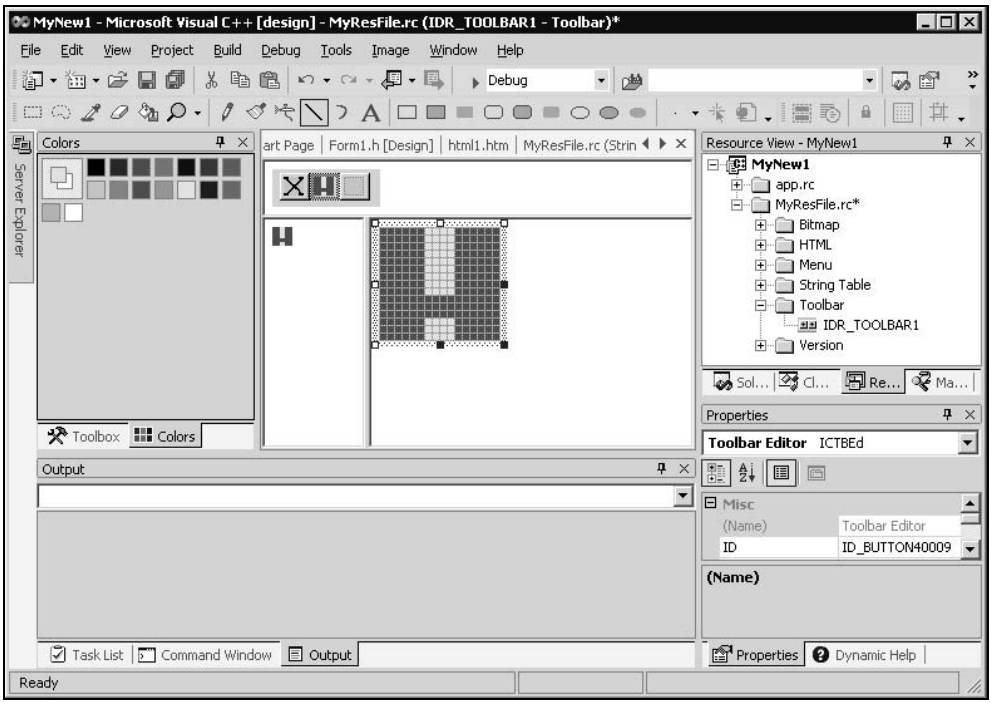


Рис. 9.12. Панель инструментов в процессе ее создания

Для каждой из создаваемых кнопок панели инструментов вы можете устанавливать значения следующих четырех свойств:

- `Height` — определяет высоту кнопки в пикселах. Смена значения для одной кнопки изменяет значения для всех кнопок панели инструментов. Рекомендованное значение — 15 пикселов;
- `ID` — определяет идентификатор кнопки;
- `Prompt` — определяет подсказку, которая будет высвечиваться при наведении указателя мыши на кнопку панели инструментов, имеющую данное свойство;
- `Width` — определяет ширину кнопки в пикселах. Рекомендованное значение — 16 пикселов.

Редактор Version Information Editor

Этот редактор позволяет просмотреть и изменить информацию о приложении (описание программы, версия файла, имя компании разработчика, авторские и торговые права, и т. д.). На рис. 9.13 показан внешний вид этого редактора.

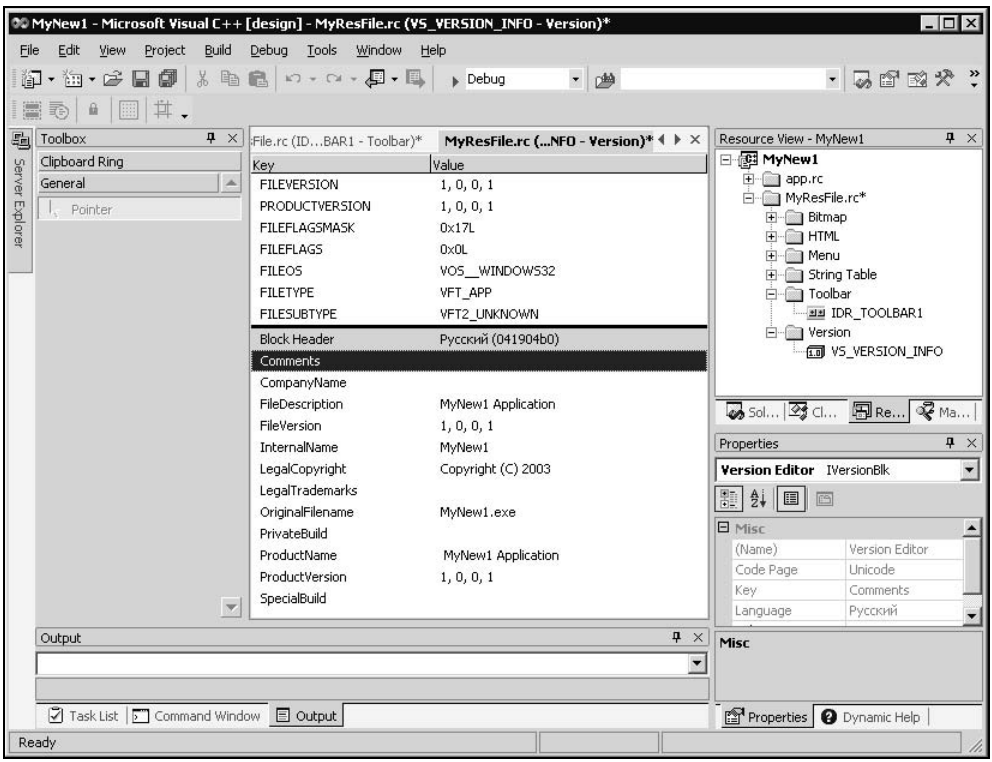
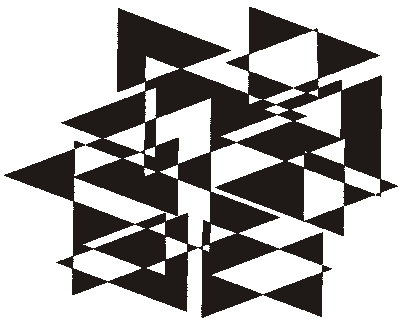


Рис. 9.13. Редактор Version Information Editor

Работа с данным редактором не представляет особой сложности, поэтому не будем на этом останавливаться.

Итак, мы рассмотрели принципы работы с файлами ресурсов в среде Visual Studio .NET 2003. На этом заканчивается *часть II*. В *части III* мы переходим к рассмотрению языков программирования C++ и C#, которые являются основными для среды разработки Visual Studio .NET 2003.

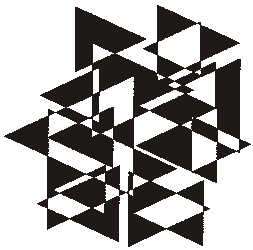
ЧАСТЬ III



Языки C++ и C#

В этой части рассмотрен синтаксис языков C++ и C#, проведен их сравнительный анализ, на основе которого показаны преимущества нового языка программирования C#. Весь материал данной части будет сопровождаться поясняющими примерами.

ГЛАВА 10



Язык C# как развитие языка C++

В этой главе мы кратко рассмотрим те особенности, которые характерны для нового языка программирования C#. Более подробно синтаксис и различие языков C# и C++ описаны в *гл. 11*.

Особенности C#

В настоящее время язык C++ является наиболее используемым языком для разработки коммерческих и бизнес-приложений. Возможности данного языка устраивают многих разработчиков, но, в действительности, не обеспечивают должной продуктивности разработки. Например, процесс написания приложения на C++ зачастую занимает значительно больше времени, чем разработка такого же приложения, скажем, на Visual Basic. Сейчас существуют языки, увеличивающие продуктивность разработки за счет потери в гибкости, которая так привычна и необходима программистам на C++. Подобные решения являются весьма неудобными для разработчиков и обычно предлагаются значительно меньшие возможности. Также, эти языки не ориентированы на взаимодействие с появляющимися сегодня системами, и очень часто они не соответствуют существующей практике программирования для Web. Многие разработчики хотели бы использовать современный язык, который позволял бы писать, читать и сопровождать программы с простотой Visual Basic, и в то же время являлся таким же мощным и гибким, как C++, обеспечивал доступ ко всем функциональным возможностям системы, взаимодействовал с существующими программами и легко работал с возникающими Web-стандартами.

Учитывая все подобные пожелания, корпорация Microsoft разработала новый язык — C#. В него входит много полезных особенностей — простота, объектная ориентированность, типовая защищенность, "сборка мусора", поддержка совместимости версий и многое другое. Данные возможности позво-

ляют быстро и легко разрабатывать приложения, особенно COM-приложения и Web-сервисы.

При создании C# его авторы учитывали достижения многих других языков программирования: C++, C, Java, SmallTalk, Delphi, Visual Basic. Надо заметить, что по причине разработки языка C++ с "чистого листа" у его авторов была возможность оставить в прошлом все неудобные и неприятные особенности (существующие, как правило, для обратной совместимости) любого из предшествующих ему языков. В результате получился действительно простой, удобный и современный язык программирования, по мощности не уступающий C++, но существенно повысивший продуктивность разработок.

Web-интеграция

Ввиду своего очень удобного объектно-ориентированного дизайна, C# является хорошим выбором для быстрого конструирования различных компонентов — от высокоуровневой бизнес-логики до системных приложений, использующих низкоуровневый код. Также следует отметить, что C# является и Web-ориентированным — используя простые встроенные конструкции языка, компоненты могут быть легко превращены в Web-сервисы, к которым можно будет обращаться из сети Интернет, посредством любого языка на любой операционной системе.

Дополнительные возможности и преимущества перед другими языками приносит в C# использование передовых Web-технологий, таких как: HTML, XML и SOAP. Среда разработки Web-сервисов позволяет программисту смотреть на существующие сегодня Web-приложения как на объекты языка C#, что дает возможность разработчикам соотнести имеющиеся Web-сервисы с их познаниями в объектно-ориентированном программировании.

Исключение ошибок

Очень часто можно прослеживать такую связь — чем более язык защищен и устойчив к ошибкам, тем меньше производительность программ, написанных на нем. К примеру, рассмотрим две крайности — очевидно, что это языки Assembler и Java. В первом случае вы можете добиться наилучшей быстроты своей программы, но вам придется очень долго "заставлять" ее работать правильно не на вашем компьютере. В случае же с Java — вы получаете защищенность, независимость от платформы, но, к сожалению, скорость программы вряд ли совместима со сложившимся представлением о скорости, например, какого-либо отдельного клиентского приложения.

Рассмотрим C++ с этой точки зрения. Соотношение в скорости и защищенности близко к желаемому результату, но практически всегда лучше понести незначительную потерю в производительности программы и приобрести такую удобную особенность, как "сборка мусора", которая не только

освобождает от утомительной обязанности управлять памятью вручную, но и помогает избежать многих потенциальных ошибок в приложении. В действительности, "сборка мусора", да и любые шаги к устранению потенциальных ошибок становятся отличительными чертами современного языка.

В C# также существуют характерные особенности для обхода возможных ошибок. Например, помимо упомянутой "сборки мусора", все переменные автоматически инициализируются средой и обладают типовой защищенностью, что позволяет избежать неопределенных ситуаций в случае, если программист забудет инициализировать переменную в объекте или попытается произвести недопустимое преобразование типов. Также в C# были предприняты меры для исключения ошибок при обновлении программного обеспечения. Изменение кода в такой ситуации может непредсказуемо изменить суть самой программы. Чтобы помочь разработчикам бороться с этой проблемой, C# включает в себя поддержку совместимости версий. В частности, в отличие от C++ и Java, если метод класса был изменен, это должно быть специально оговорено. Это позволяет обойти ошибки в коде и обеспечить гибкую совместимость версий. Также новой особенностью является встроенная поддержка интерфейсов и наследования интерфейсов. Данные возможности позволяют разрабатывать сложные системы и развивать их со временем.

Простота использования

Важной и отличительной особенностью C# от C++ является его простота. Например, всегда ли вы помните, когда пишете программу на C++, где нужно использовать `->`, где `::`, а где `.`? Даже если нет, то компилятор всегда поправляет в случае ошибки. Это говорит лишь о том, что в действительности можно обойтись только одним оператором, а компилятор сам будет распознавать его значение.

Так, в C# оператор `->` используется очень ограниченно (в блоках `unsafe`, о которых речь пойдет далее), оператор `::` вообще не существует. Практически всегда используется только оператор `.` и больше не нужно стоять перед выбором.

Еще один пример. При написании программ на C/C++ приходилось думать не только о типах данных, но и об их размерах в конкретной реализации. В C# все упрощено — теперь символ Unicode называется просто `char` (а не `wchar_t`, как в C++) и 64-битное целое теперь — `long` (а не `_int64`). Также в C# нет знаковых и беззнаковых символьных типов.

В C#, так же как и в Visual Basic, после каждого выражения `case` в блоке `switch` подразумевается оператор `break`. И не будет происходить "странных вещей", если вы забыли поставить этот `break`. Однако, если вы действительно хотите, чтобы после одного выражения `case` программа перешла к сле-

дующему выражению, можно переписать программу с использованием, например, оператора `goto`.

Еще одно упрощение — в C# не существует множественного наследования классов. В действительности его использование является вовсе не самой простой задачей и зачастую приводит к ошибкам. Вместо этого, в C# принята полная поддержка концепций COM+, которые исключают необходимость использования множественного наследования.

Программистам не так легко во время изучения C++ полностью освоиться с механизмом ссылок и указателей. Многие путаются в использовании операторов `*` и `&`. В C# нет указателей. В действительности, неоднозначность указателей соответствовала их полезности. Например, трудно себе представить программирование без указателей на функции. В соответствии с этим в C# присутствуют *делегаты* (Delegates) — как прямые аналоги указателей на функции, но их отличает типовая защищенность, безопасность и полное соответствие концепциям объектно-ориентированного программирования. Пример использования делегатов приведен в листинге 10.1.

Листинг 10.1. Использование делегатов

```
delegate void SimpleDelegate();
  //  объявление типа Delegate, принимаемых значений и возвращаемого типа
class Test
{
   static void F()
   {
      System.Console.WriteLine("Test.F");
   }
   static void Main()
   {
     //  создание объекта делегата
     SimpleDelegate d = new SimpleDelegate(F);
     d();    //  запуск делегата
   }
}
```

Делегаты — это объекты, производные от общего базового объекта `System.Delegate`. Этот тип представляет собой некую сущность, которую можно вызывать.

В случае с экземпляром объекта — сущность состоит из экземпляра и его метода, а в случае статического метода — из класса и статического метода данного класса. Именно благодаря этому не нарушаются концепции объектно-ориентированного программирования, а точнее, не используются методы объекта в отрыве от самого объекта.

Типовая защищенность

Типовая защищенность способствует написанию устойчивых программ. В C# все динамические объекты и массивы автоматически инициализируются нулем. Несмотря на то, что C# автоматически не инициализирует локальные переменные, компилятор выдаст предупреждение, если вы попытаетесь использовать их до инициализации. Кроме того, при обращении к массиву выполняется проверка на выход за его границы. Таким образом, в C#, в отличие от C++, невозможно изменить не отведенную для вас память.

В C# нельзя создать ссылку на произвольную область памяти. Все приведения типов должны быть безопасными. Например, вы не можете использовать приведение между типами `reference` (обращение по указателю на область памяти) и `value` (обращение к значению). "Сборка мусора" обеспечит отсутствие в коде пустых ссылок.

Типовая защищенность неразрывно связана с контролем на переполнение. Не допускаются арифметические операции, которые будут переполнять допустимое значение переменной или объекта. Хотя, конечно, существует лишь неполный набор факторов, которые говорят о явном переполнении переменной. Если же не нравится такая проверка — можно ее отменить.

Удобство и современность C#

Хотелось бы подчеркнуть такие качества C#, как удобство и современность. Когда вы начнете работу с C#, то увидите, что довольно большое значение в нем имеют пространства имен. Так в C#, помимо выражения `using`, предоставляется еще одна очень удобная возможность — использование дополнительного имени (`alias`) пространства имен или класса.

Современность C# проявляется и в новых шагах к облегчению процесса отладки программы. Стандартным средством для отладки программ на стадии разработки в C++ является маркировка обширных частей кода директивами `#ifdef` и другими. В C#, используя атрибуты, ориентированные на условные слова, вы можете гораздо быстрее написать отлаживаемый код, текст которого приведен в листинге 10.2.

Листинг 10.2. Создание отлаживаемого кода

```
  //  код из первого файла
using System;
namespace DotSiteRes
{
   public class HelloWorld
   {
      [conditional("DEBUG")]
```

```
      //  последующий метод будет выполняться только в случае
      //  определения DEBUG
      public static void SayHello()
      {
         Console.WriteLine("Hello, World!");
         return;
      }
...
   }
}

  //  код из второго файла
using System;
using DotSiteRes;

#define DEBUG

class CallingDotSiteRes
{
   public static void Main(string[] args)
   {
      HelloWorld.SayHello();
      return;
   }
}
```

В результате выполнения программы на экран будет выведена строка `Hello, World!`. Без определения `DEBUG` листинг не исполнится. Не менее полезное использование атрибутов будет продемонстрировано далее.

В наше время, когда усиливается связь между миром коммерции и миром разработки программного обеспечения, и компании тратят много усилий на планирование бизнеса, ощущается необходимость в соответствии абстрактных бизнес-процессов их программным реализациям. К сожалению, большинство языков реально не имеют прямого пути для связи бизнес-логики и кода. Например, сегодня многие программисты комментируют свои программы для объяснения того, какие классы реализуют какой-либо абстрактный бизнес-объект.

C# позволяет использовать типизированные, расширяемые метаданные, которые могут быть прикреплены к объекту. Архитектурой проекта могут определяться локальные атрибуты, которые будут связаны с любыми элементами языка — классами, интерфейсами и т. д. Разработчик может программно проверить атрибуты какого-либо элемента. Это существенно упрощает работу, например, вместо того чтобы писать автоматизированный

инструмент, который будет проверять каждый класс или интерфейс, на то, является ли он действительно частью абстрактного бизнес-объекта, можно просто воспользоваться сообщениями, основанными на определенных в объекте локальных атрибутах. Следующий пример демонстрирует это (текст приведен в листинге 10.3).

Листинг 10.3. Использование атрибутов

```
using System;
[AttributeUsage(AttributeTargets.All)]

  //  пример использования атрибутов, с их помощью говорится,
  //  где может использоваться атрибут-класс, определяемый ниже
public class HelpAttribute: Attribute
{
   //  атрибут-класс — наследник класса System.Attribute
   //  в HelpAttribute, Help является непосредственным именем
   //  атрибута-класса, а Attribute — суффиксом
   //  при использовании атрибут-класса HelpAttribute, можно
   //  обращаться к нему просто Help

   public HelpAttribute(string url)
   {
      this.url = url;
   }
   public string Topic = null;
   private string url;
   public string Url
   {
      get
      {
         return url;
      }
   }
}

[Help("http://www.mycompany.com/:/Class1.htm")]
  //  применение атрибута к новому классу
public class Class1
{
   [Help("http://www.mycompany.com/:/Class1.htm", Topic = "F")]
   //  применение атрибута к методу
   public void F() {}
}
```

```
class Test
{
   //  класс, проверяющий атрибуты
   static void Main()
   {
      type MyType = typeof(Class1);
      object[] arr =
      MyType.GetCustomAttributes(typeof(HelpAttribute));
      //  чтение атрибутов класса Class1
      if (arr.Length == 0)
      Console.WriteLine("Class1 не имеет атрибута Help.");
      else
      {
         HelpAttribute ha = (HelpAttribute) arr[0];
         //  вывод атрибутов
         Console.WriteLine("Url = {0}, Topic = {1}",
         ha.Url, ha.Topic);
      }
   }
}
```

Дополнительные возможности

Реальный опыт разработки свидетельствует о том, что некоторым программам и по сей день требуется встроенный код либо из соображений производительности, либо для взаимодействия с существующими API. Эта причина может заставить разработчиков использовать C++, даже когда они предпочли бы более продуктивную среду разработки.

В C# для устранения этой проблемы были приняты несколько модификаций: включена встроенная поддержка COM-технологии и WinAPI, допускается ограниченно использовать встроенные указатели. Разработчику больше не нужно точно реализовывать `IUnknown` и другие COM-интерфейсы. Теперь эти особенности являются встроенными.

Также программы на C# могут использовать существующие COM-объекты независимо от того, на каком языке они были написаны. Для тех разработчиков, которым это необходимо, в C# была включена специальная функция, которая позволяет программе обращаться к любому существующему API. Внутри помеченного блока (`unsafe`) программисту позволяется использовать указатели и традиционные особенности C++, такие как управление памятью вручную и арифметика указателей. Это отличает C# от других сред разработки. Вместо того, чтобы отказаться от уже существующего кода C++, вы можете разрабатывать C#-приложения с его использованием. В листинге 10.4 приведен текст программы с применением блоков.

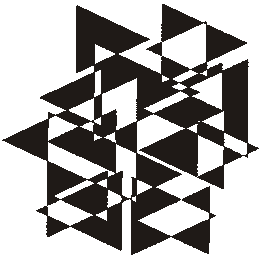
Листинг 10.4. Использование блоков `unsafe`

```
struct Point
{
   public int x;
   public int y;
   public override string ToString()
   {
      return "(" + x + "," + y + ")";
   }
}

class Test
{
   unsafe static void Main()
   {
      Point point;
      Point* p = &point;
      p->x = 10;
      p->y = 20;
      Console.WriteLine(p->ToString());
   }
}
```

Как вы видите, внутри блоков `unsafe`, как и в C++, возможно применение указателей. Также допускается и использование оператора `->`, который может быть заменен на `(*p)`.

Итак, мы кратко рассмотрели основные и наиболее яркие особенности языка C# в сравнении с C++. В *гл. 11* и *12* будут подробно рассмотрены синтаксисы этих языков программирования.

ГЛАВА 11



Синтаксис языков C# и C++

В данной главе рассмотрим синтаксис языка C# и отметим его основные отличия от синтаксиса языка C++. Так как в основном синтаксис обоих языков является практически одинаковым, в целях экономии места в книге мы не будем рассматривать их отдельно.

Основы языка C#

В этом разделе рассмотрим простое консольное приложение на C#. Затем изучим типы данных, предоставляемые языком, и сравним их с типами данных языка C++. Кроме того, ознакомимся со способами преобразования одних типов данных в другие. Изучим особенности работы с массивами. Завершит настоящий раздел рассмотрение методов организации ввода и вывода в консольных приложениях.

Простая программа на C#

В *гл. 2* были приведены примеры простых консольных приложений на разных .NET-языках, но не акцентировалось внимание на отличиях в синтаксисе. В листинге 11.1 приведен код простой консольной программы, которая выводит строку текста на экран.

Листинг 11.1. Простое консольное приложение на C#

```
Using System;

  //  класс запуска приложения
public class MyConsoleApplication
{
   //  метод Main выполняется первым при запуске приложения
   public static void Main()
```

```
    {
        //  вывод строки текста на экран
        System.Console.WriteLine("Эта строка будет напечатана!");
    }
}
```

Обратите внимание на тот факт, что метод `Main()` в C# пишется с *заглавной буквы*, в отличие от C++. Кроме того, данный метод содержится *внутри класса*, а не существует сам по себе.

Примечание

Как и C++, язык C# является регистрозависимым.

В языке C# классы используются для объявления или описания объектов. Они описывают атрибуты и поведение объектов. Классы, таким образом, служат своеобразными описаниями, а объекты являются реальными образованиями, которые существуют во время выполнения программы.

Комментарии

В языке C# имеется три типа синтаксиса комментариев:

- однострочный комментарий;
- многострочный комментарий;
- документирующий XML-комментарий.

Рассмотрим их по порядку.

Однострочный комментарий может начинаться с любой позиции любой строки кода программы. Он начинается с двух символов слэша (`//`):

```
System.Console.WriteLine("Привет!");   //  вывод на экран слова Привет!
```

Многострочный комментарий может содержать одну или несколько строк текста внутри ограничителей начала (`/*`) и конца (`*/`) многострочного комментария:

```
/*
Это многострочный
комментарий
*/
```

Третий тип комментариев — *документирующий XML-комментарий* не поддерживается языком C++. Такой комментарий начинается с трех символов слэша (`///`). Компилятор языка C# обладает возможностью чтения содержимого XML-комментариев и генерирует из этих комментариев XML-документацию для данного приложения. Такую документацию можно извлечь

в отдельный XML-файл, а затем применить к нему таблицы стилей XML для просмотра содержимого файла в браузере.

Для того чтобы компилятор языка разобрался в содержимом XML-комментариев, используются специальные теги, которые носят название *тегов XML-документации*. Они приведены в табл. 11.1.

Таблица 11.1. *Теги XML-документации*

Тег	Краткое описание
`<c>`	Позволяет выделить текст в виде кода внутри описания метода или элемента
`<code>`	Позволяет выделить несколько строк текста в виде кода внутри описания метода или элемента
`<example>`	Позволяет привести пример использования метода или другого элемента кода
`<exception>`	Позволяет указывать, какие исключения могут возникнуть при работе данного класса
`<include>`	Позволяет ссылаться на внешний файл, который описывает исходный код
`<list>`	Позволяет создать список текстовых значений
`<para>`	Используется внутри тегов (например, `remarks` или `returns`) и позволяет структурировать текст
`<param>`	Позволяет описывать параметры конкретного метода
`<paramref>`	Позволяет указать, что данное слово является параметром
`<permission>`	Позволяет указать способ доступа к данному методу
`<remarks>`	Позволяет дать общее описание класса или типа
`<returns>`	Позволяет описать возвращаемое данным методом значение
`<see>`	Позволяет определять ссылки внутри текста
`<seealso>`	Позволяет определять текст, который будет отображаться в секции **Смотри также** (See Also) внутри XML-документации
`<summary>`	Позволяет дать описание какому-либо элементу
`<value>`	Позволяет дать описание свойства

Рассмотрим простой пример использования некоторых тегов XML-документации для простого консольного приложения (листинг 11.2).

Листинг 11.2. Теги XML-документации в консольном приложении

```
  ///  описание класса MyClass
public class MyClass
{
   ///  <summary>MyMethod — это метод класса MyClass.
   ///  <para>Так вы можете создать второй параграф в описании.
        <seecref="System.Console.WriteLine"/> для информации о выходных
        значениях.</para>
   ///  <seealso cref="MyClass.Main"/>
   ///  </summary>
   public static void MyMethod(int a)
   {
      a = 10;
   }

   ///  описание метода Main
   ///  <summary>
   ///  выводит на экран выражение Привет!
   ///  </summary>
   public static void Main ()
   {
      //  вывод строки текста на экран
      System.Console.WriteLine("Привет!");
   }
}
```

Такие комментарии могут использоваться для регулярного обновления документации. В работе над кодом можно легко обновить комментарии и сгенерировать результирующий файл в формате XML.

Для извлечения комментариев из файла class1.cs и создания XML-документа с именем class1.xml вы можете воспользоваться опцией `/doc` компилятора csc.exe таким образом:

```
csc /doc:class1.xml class1.cs
```

После успешной компиляции образуется XML-файл, текст которого приведен в листинге 11.3.

Листинг 11.3. Файл class1.xml

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>Class1</name>
```

```
   </assembly>
   <members>
      <member name="T:MyClass">
   описание класса MyClass
   </member>
   <member name="M:MyClass.MyMethod(System.Int32)">
      <summary>MyMethod — это метод класса MyClass.
      <para>Так вы можете создать второй параграф в описании.
      <see cref="M:System.Console.WriteLine"/> для информации
      о выходных значениях.</para>
      <seealso cref="M:MyClass.Main"/>
      </summary>
   </member>
   <member name="M:MyClass.Main">
      описание метода Main
      <summary>
      выводит на экран выражение Привет!
      </summary>
   </member>
   </members>
</doc>
```

Согласитесь, что это достаточно удобный способ генерирования документации.

Типы данных

Язык C#, как и C++, относится к строго типизированным языкам. То есть, компилятор и среда времени выполнения строго следят за соответствием типов данных в выражениях. У каждой переменной должен быть обязательно объявлен ее тип. Синтаксис описания переменных всегда строится следующим образом:

```
Тип_переменной Название_переменной [начальное_значение_переменной]
```

Сначала указывается тип переменной, затем ее название. Необязательным является указание начального значения для данной переменной.

Все типы данных можно условно разделить на *простые*, *структурные*, *ссылочные*, *строковые* и *типы перечисления*. Рассмотрим эти типы данных применительно к языку C# и отметим те отличия, которые присутствуют в языке C++.

Простые типы данных

К простым типам данных относятся *логические* и *числовые* типы. Числовые, в свою очередь, делятся на *целочисленные* типы и типы *с плавающей точкой*.

Логический (булевый) *тип данных* обозначается ключевым словом `bool`. Может принимать одно из двух возможных значений: `true` или `false`. Пример объявления переменной логического типа:

```
bool MyBool = false;
```

Отметим, что в языке C# единственными допустимыми значениями для типа `bool` являются `true` и `false`. Целые значения `1`, `0` и `-1` не допустимы. Заметим также, что, в отличие от языка C++, в C# невозможно выполнить преобразование между типами `bool` и `int`.

В состав *целочисленных* типов данных входят восемь числовых типов и один символьный: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` и `char`. Все целочисленные типы (кроме символьного `char`) могут быть как со знаком, так и без него.

В табл. 11.2 приведены характеристики целочисленных типов данных языка C#.

Таблица 11.2. *Характеристики целочисленных типов данных языка C#*

Тип данных	Размер (в разрядах)	Диапазон допустимых значений
`sbyte`	8-битное целое со знаком	–128—127
`byte`	8-битное целое без знака	0—255
`char`	16-битный символ Unicode	0000—FFFF
`short`	16-битное целое со знаком	–32 768—32 767
`ushort`	16-битное целое без знака	0—65 535
`int`	32-битное целое со знаком	–2 147 483 648—2 147 483 647
`uint`	32-битное целое без знака	0—4 294 967 295
`long`	64-битное целое со знаком	–9 223 372 036 854 775 808— 9 223 372 036 854 775 807
`ulong`	64-битное целое без знака	0—18 446 744 073 709 551 615

Обратите внимание на тот факт, что в языке C++ нет эквивалента типу `byte`. Но его можно заменить типом `char`. Кроме того, в языке C++ тип `long` имеет 32 разряда, а в C# — 64.

Отметим также, что тип `char` в языках C# и C++ отличается. Так, в C++ он имеет 8 разрядов, а в C# — 16 разрядов. Таким образом, `char` в C# является аналогом типа `wchar_t` языка C++.

Примеры объявления переменных различных целочисленных типов могут быть следующими:

```
byte x = 12;
sbyte y = -6;
int z;
```

Так как тип `char` предназначен для хранения одного символа в формате Unicode, он может быть представлен в виде четырех шестнадцатеричных цифр, с префиксом \u или \U. При необходимости, число дополняется слева незначащими нулями для того, чтобы его длина составляла четыре цифры. Кроме того, значение типа `char` может быть указано в шестнадцатеричной форме с добавлением префикса \x. Рассмотрим примеры объявления переменных типа `char`:

```
char a;
char b = 'W';
char c = '\u023C';
char d = '\x000D'
```

Существуют специальные управляющие последовательности, которые представляют собой специальные символы. Допустимые управляющие последовательности языка C# представлены в табл. 11.3.

Таблица 11.3. *Допустимые управляющие последовательности языка C#*

Управляющая последовательность	Краткое описание
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратный слэш
\0	Нуль (Null)
\a	Звуковой сигнал (Bell)
\b	Стирание символа слева (Back Space)
\f	Подача страницы (Form Feed)
\n	Перевод каретки или пропуск строки
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция

Типы данных с плавающей точкой, имеющиеся как в языке C#, так и C++, — это `float` и `double`.

Тип данных `float` может принимать любое значение в диапазоне от 1.5×10^{-45} до 3.4×10^{38}. Точность типа данных составляет семь знаков.

Значение типа `float` может быть записано как в обычной форме, так и в экспоненциальной:

```
float a = 272.489;
float b = 2.332E-10;
float c = 5.325e5;
```

Тип данных `double` может принимать любое значение в диапазоне от 5.0×10^{-324} до 1.7×10^{308}. Точность типа данных составляет пятнадцать — шестнадцать знаков. Значение типа `double` может быть также записано как в обычной, так и в экспоненциальной форме:

```
double a = 39423.2878492;
double b = 2.346E27;
```

В языке C# имеется еще один тип данных с плавающей точкой — `decimal`. Он отсутствует в языке C++. Этот тип данных может принимать любое значение в диапазоне от 1.0×10^{-28} до 7.9×10^{28}. Точность типа `decimal` составляет двадцать восемь — двадцать девять знаков. Можно заметить, что большая точность предоставляется в ущерб диапазону допустимых значений.

Структурные типы данных

Структурные типы данных могут включать в себя целочисленные типы, типы данных с плавающей точкой, а также некоторые определенные программистом типы данных. Структурные типы данных являются своеобразными контейнерами, которые могут хранить в себе набор разнотипных элементов. Для объявления структуры используется ключевое слово `struct`.

Рассмотрим пример объявления структурного типа данных на языке C# (листинг 11.4).

Листинг 11.4. Объявление и работа со структурным типом данных

```
  //  объявление и инициализация структуры
using System;
public struct PointXYZ
{
   public int x, y, z;
   public PointXYZ(int p1, int p2, int p3)
   {
      x = p1;
      y = p2;
      z = p3;
   }
}
```

```
class Class1
{
   public static void Main()
   {
      //  инициализация экземпляров
      PointXYZ myPoint1 = new PointXYZ();
      PointXYZ myPoint2 = new PointXYZ(5, 10, 7);

      //  вывод сообщения на экран
      Console.Write("Первая точка: ");
      Console.WriteLine("x = {0}, y = {1}, z = {2}",
      myPoint1.x, myPoint1.y, myPoint1.z);
      Console.Write("Вторая точка: ");
      Console.WriteLine("x = {0}, y = {1}, z = {2}", myPoint2.x,
      myPoint2.y, myPoint2.z);
   }
}
```

В результате выполнения данного примера вы увидите на экране следующее:

```
Первая точка: x = 0, y = 0, z = 0
Вторая точка: x = 5, y = 10, z = 7
```

Ссылочные типы данных

Ссылочные типы данных — это библиотечные или пользовательские типы, которые размещаются в памяти *"кучи"* (heap). Это значит, что ссылочные типы данных используют дополнительные ресурсы системы. Такие типы данных управляются встроенным "сборщиком мусора" (garbage collector), которого нет в языке C++. В языке C# имеется четыре ссылочных типа данных.

1. *Классы* — могут содержать в себе множество различных членов языка C#, описывают уникальные типы данных.
2. *Интерфейсы* — обеспечивают взаимодействие с открытыми атрибутами и поведением классов.
3. *Делегаты* — позволяют динамически обращаться к методам класса. При этом никогда не возникает конфликт типов. Используются в случаях, когда реализуемый метод не известен до выполнения программы.
4. *Массивы* — предназначены для хранения множества элементов определенного типа данных.

Строковый тип данных

Как в языке C++, так и в C# строковый тип данных представлен типом `string`. Значение переменной такого типа представляется в виде строки символов Unicode. Данный тип может включать в себя любой допустимый набор символов, в том числе и управляющие последовательности (см. табл. 11.3). Строковое значение заключается в двойные кавычки. Ниже приведены примеры объявления строковых переменных:

```
string a = "Visual C++";
string b = "Visual C# \n Visual C++";
```

Отметим, что в языке C++ строковый тип `string` первоначально совпадал с простой строкой языка C и представлял собой указатель на массив символов, который завершался нулевым значением. По спецификации Standard C++ данный тип ссылается на строковый тип *STL* (Standard Template Library, стандартная библиотека шаблонов).

В языке C# тип `string` является встроенным и его представление скрыто от пользователя.

Кроме того, в языке C# имеется еще один тип строковых литералов, который носит название *"буквального строкового литерала"* (verbatim string literal). Он отличается от обычного строкового литерала тем, что к строке добавляется префикс @. В данном типе строковых литералов управляющие последовательности не обрабатываются, а воспринимаются компилятором как простые символы. Это удобно для игнорирования управляющих последовательностей в именах файлов и других случаях. Примеры объявления буквальных строковых литералов такие:

```
string a = @"Visual C++ \n Visual C#";
string b = @"Это строка, которая занимает
несколько строк,
т. к. управляющие символы не воспринимаются";
```

Типы перечисления

В обоих языках (C++ и C#) для объявления *типов перечисления* используется ключевое слово `enum`. Тип `enum` представляет собой список значений-констант.

Элементам перечисления в языке C++ может присваиваться только тип `int`, а в языке C# — `byte`, `int`, `long` и `short`.

Рассмотрим простой пример объявления типа перечисления (или перечисляемого типа):

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

В данном случае (как и во всех остальных случаях, когда элементам перечисления не присвоены значения) значение первого элемента `Monday` будет равно нулю, а значение каждого последующего элемента будет увеличиваться на единицу. Таким образом, значение последнего элемента `Sunday` будет равно шести.

В следующем примере значение первого элемента будет равно единице, а двух последних 11 и 12:

```
enum Days {Monday = 1, Tuesday, Wednesday, Thursday, Friday,
Saturday = 11, Sunday};
```

Так как элементы перечисления в языке C# могут принадлежать к различным типам, вы можете указать нужный тип после имени перечисления через двоеточие:

```
enum Days: short {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday};
```

Определенность значений

Определенность значений — это правило, по которому каждая переменная должна получить какое-либо начальное значение, прежде чем программа обратится к данной переменной.

Процесс получения переменной начального значения носит название *инициализации переменной*. Неинициализированные переменные называются *неопределенными переменными*.

Если вы пишете программу на языке C++, то можете использовать неопределенные переменные. Компилятор языка C# выявляет неопределенные переменные на этапе компиляции программы и выдает сообщение об ошибке.

Правила инициализации переменных зависят от того, в каком месте программы находится переменная. Если переменная является *локальной* (объявлена внутри метода или какого-либо блока), то такая переменная изначально является не инициализированной.

Если переменная объявлена как член класса (и является, таким образом, *переменной класса*), то она инициализируется путем присваивания стандартных значений (если в коде программы нет явной инициализации переменной).

В табл. 11.4 приведены значения, которые устанавливаются компилятором по умолчанию для разных типов данных (для языка C#).

Таблица 11.4. *Значения по умолчанию для переменных разных типов данных*

Тип данных	Значение по умолчанию	Тип данных	Значение по умолчанию
bool	false	byte	0
char	0000	decimal	0.0
double	0.0	float	0.0
int	0	long	0
sbyte	0	short	0
uint	0	ulong	0
ushort	0		

Преобразование типов данных

Так как иногда возникает необходимость преобразовать один тип данных в другой, в языках C++ и C# имеются механизмы преобразования типов.

Существует два вида преобразований типов данных:

- *неявные преобразования* (implicit conversion) — происходят без вмешательства программиста;
- *явные преобразования* (explicit conversion) — выполняются с помощью *операции приведения типа* (cast) к выражению. Такие преобразования применяются в тех случаях, когда в результате может произойти потеря данных или ошибка.

В табл. 11.5 приведены допустимые неявные преобразования для типов данных языка C# (для C++ почти все в данной таблице тоже подойдет, за исключением типов данных, отсутствующих в языке).

Таблица 11.5. *Допустимые неявные преобразования типов данных*

Тип данных	Типы данных, в которые может осуществляться преобразование
bool	Нет
byte	short, ushort, int, uint, long, ulong, float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
decimal	Нет
double	Нет
float	double

Таблица 11.5 *(окончание)*

Тип данных	Типы данных, в которые может осуществляться преобразование
int	long, float, double, decimal
long	float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
uint	long, ulong, float, double, decimal
ulong	float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal

Кроме приведенных в табл. 11.5 допустимых неявных преобразований, возможно неявное преобразование целочисленного нулевого литерала (0) в тип enum.

Неявное преобразование типов осуществляется автоматически без какого-либо специального синтаксиса.

Для явного преобразования типов используются следующие синтаксисы:

```
Простой_тип_данных (переменная)
```

или:

```
(Простой_тип_данных) переменная
```

Рассмотрим несколько примеров неявного и явного преобразования типов данных:

```
  //  пример неявного преобразования типов
short a = 1;
int b = a;
  //  пример явного преобразования типов
int c = 7;
float d;
d = float( c);
  //  другой синтаксис явного преобразования типов
int e = 7;
float f;
f = (float) e;
```

Работа с массивами

В языке C++ массив представляет собой указатель на непрерывный блок памяти, который управляется индексами или указателями для записи, хра-

нения и извлечения данных. В языке C# массивы представляют собой объекты со встроенными функциями различных операций (эти операции мы рассмотрим далее).

Объявление массивов в языке C# несколько отличается (синтаксически) от объявления массивов в языке C++. Квадратные скобки при объявлении массива в языке C++ располагаются после идентификатора, а в C# — после указания типа.

Все массивы можно условно разделить на *одномерные*, *n-мерные* и *свободные*.

Одномерные массивы

Одномерные массивы предназначены для хранения и работы со списком элементов. Все элементы должны быть одного типа (например, `int`).

Синтаксис объявления одномерного массива в языке C++ выглядит так:

```
Тип_массива идентификатор_массива[]
```

В языке C# синтаксис следующий:

```
Тип_массива[] идентификатор_массива
```

Рассмотрим примеры объявления одномерных массивов на языках C++ и C#:

```
  //  C++
int a1[10];  //  одномерный массив на 10 элементов
  //  C#
int[] a1;    //  объявление массива без инициализации
```

Массивы могут объявляться без инициализации. При инициализации массива необходимо указать значение типа `int` в квадратных скобках, как показано в следующем примере:

```
  //  массив из двух строк (язык C#)
string[] MyString = new string[2] {"первое значение", "второе значение"};
```

N-мерные массивы

Оба языка поддерживают объявление многомерных массивов. Процедура похожа на объявление одномерных массивов. Для инициализации n-мерных массивов необходимо указать целочисленное значение размерности для каждого измерения. Любое измерение добавляет запятую в квадратные скобки. Рассмотрим пример объявления двумерного массива:

```
  //  язык C#, объявление двумерного массива 10x10 элементов
decimal[, ] MyArray = new decimal[10, 10];
```

Свободные массивы

Свободные массивы (jagged arrays) — это многомерные массивы, в которых элементы разных измерений могут иметь неодинаковые размеры. Это очень удобно, когда приложения не охватывают весь диапазон возможных значений. Рассмотрим пример объявления свободного массива, в котором все элементы имеют различный размер:

```
  //  объявление свободного массива, состоящего из двух элементов
int[][] myArray = new int[][];

  //  инициализация элементов массива
myArray[0] = new int[5] {1, 2, 3, 4, 5};
myArray[1] = new int[4] {6, 7, 8, 9};
```

Операции с массивами

Язык C# предоставляет программисту намного больше встроенных возможностей для работы с массивами, нежели язык C++. Так как массивы в C++ являются простыми указателями на блоки памяти с однотипными данными, вся обработка таких массивов "ложится на плечи" программиста (для любой задачи, например сортировки массива, приходится писать собственный код). Язык C# позволяет обращаться к свойствам и методам массивов по аналогии обращения к свойствам и методам других объектов. Эти свойства и методы и их краткое описание приведены в табл. 11.6.

Таблица 11.6. *Свойства и методы массивов языка C#*

Метод или свойство	Краткое описание
`AsList`	Возвращает значения, входящие в массив в виде списка
`BinarySearch`	Находит значение в одномерном массиве с помощью метода двоичного поиска
`Clear`	Сбрасывает в `0` или `null` заданный диапазон элементов массива
`Clone`	Возвращает ограниченную копию массива (массив со ссылками на объекты, вместо самих объектов)
`Copy`	Копирует заданный диапазон элементов массива в другой массив
`CopyTo`	Копирует все данные из одного одномерного массива в другой одномерный массив
`CreateInstance`	Создает новый экземпляр данного массива

Таблица 11.6 *(окончание)*

Метод или свойство	Краткое описание
`Equals`	Сравнивает два массива
`GetEnumerator`	Возвращает интерфейс `IEnumerator` для одномерного массива
`GetHashCode`	Возвращает уникальный идентификатор массива
`GetLength`	Возвращает количество элементов массива в заданном измерении
`GetLowerBound`	Возвращает нижнюю границу измерения массива
`GetType`	Возвращает тип данных массива
`GetUpperBound`	Возвращает верхнюю границу измерения массива
`GetValue`	Возвращает значение указанного элемента массива
`IndexOf`	Находит первое вхождение заданного значения и возвращает его индекс (порядковый номер)
`Initialize`	Инициализирует каждый элемент массива значением по умолчанию
`LastIndexOf`	Находит последнее вхождение заданного значения и возвращает его индекс
`Reverse`	Упорядочивает элементы одномерного массива в обратном порядке
`Sort`	Выполняет сортировку элементов одномерного массива
`IsReadOnly`	Возвращает значение `true`, если массив предназначен только для чтения
`IsSynchronized`	Возвращает значение `true`, если массив синхронизирован
`Length`	Возвращает количество элементов массива во всех его измерениях
`Rank`	Возвращает число, равное размерности массива
`SetValue`	Устанавливает значение для указанного элемента массива
`SyncRoot`	Возвращает объект синхронизации массива
`ToString`	Возвращает строковое представление массива

Приведу простой пример, в котором при помощи свойства `Length` получено значение, равное количеству элементов массива (листинг 11.5).

Листинг 11.5. Работа со свойством массива

```
using System;
class Class1
{
public static void Main()
   {
      string[] a = new string[10];
      Console.WriteLine("Число элементов массива a — {0}", a.Length);
   }
}
```

В результате выполнения программы вы увидите на экране такую строку:

```
Число элементов массива a — 10
```

Ввод и вывод в консольных приложениях

Рассмотрим два примера на языке C#, которые наглядно демонстрируют способы взаимодействия консольного приложения с пользователем (листинги 11.6 и 11.7).

Листинг 11.6. Пример № 1

```
using System;
class Class1
{
public static void Main()
   {
      string s;  //  переменная для ввода строки
      //  запрос ввода строки
      Console.Write("Введите произвольную строку текста: ");
      //  прием данных от пользователя
      s = Console.ReadLine();
      //  вывод введенной пользователем строки
      Console.WriteLine("Вы ввели строку: {0}", s);
   }
}
```

Листинг 11.7. Пример № 2

```
using System;
class Class1
{
public static void Main(string[] args)
```

```
    {
        Console.WriteLine("Аргумент, переданный программе: {0}", args[0]);
        Console.ReadLine();
    }
}
```

В первом примере (листинг 11.6) программа запрашивает строку текста, и после того как пользователь ее вводит, происходит отображение строки. В этом примере используются стандартные методы класса `Console` — `Write()`, `WriteLine()` и `ReadLine()`. При этом, в последнем вызове метода `WriteLine()` внутри строкового аргумента "`Вы ввели строку: `" указан параметр `{0}`. Вторым аргументом метода служит переменная `s`, в которой хранится введенная пользователем строка. При вызове метода параметр `{0}` замещается значением переменной `s`.

На рис. 11.1 показана работа данного консольного приложения.

Рис. 11.1. Работа примера № 1

Второй пример (см. листинг 11.7) демонстрирует возможность передачи данных приложению с помощью параметра командной строки. Если вы запустите пример № 2 из командной строки с параметром, например, так:

```
ConsoleApplication12.exe C#
```

то увидите на экране следующий результат (рис. 11.2).

В данном примере метод `Main()` указан с параметром. Этот параметр представляет собой массив с именем `args`. Массив может содержать в себе список данных типа `string`. Таким образом, параметр `{0}` при выводе текста на

экран будет замещен значением элемента `args[0]`. Если вы хотите ввести несколько значений в командной строке, то они будут находиться в следующих элементах массива: `args[1]`, `args[2]` и т. д.

Рис. 11.2. Работа примера № 2

Отметим, что в языке C++ параметры командной строки передаются методу `main()` с помощью двух переменных:

- `argc` — счетчика введенных переменных;
- `argv` — массива указателей на строки, содержащие параметры командной строки.

Операции и операторы

В данном разделе мы рассмотрим основные операции языков C++ и C#, а также операторы, которые имеются в этих языках. Как и ранее, я буду рассказывать об операциях и операторах языка C#, т. к. они являются более функциональными. А о различиях с языком C++ буду упоминать в ходе изложения материала.

Выражения и операции

Языки C# и C++ предоставляют программисту полный набор языковых элементов для написания выражений.

Выражение — это конструкция языка, которая содержит в себе данные, операции, и имеется строгий порядок выполнения этих операций.

В общем случае в состав выражения входят:

- *операнды* — данные, над которыми производятся действия. Операндами могут быть переменные (в т. ч. массивов), константы и другие языковые элементы;
- *знаки операций* — это знаки, определяющие операцию, которая должна быть произведена над операндами;
- *скобки* — это символы круглых скобок `()`, которые служат для указывания порядка выполнения операций.

Далее рассмотрим выражения, которые образованы с помощью встроенных операций языка C# (отличия от C++ также упомянем).

Все операции языка C# можно разбить на три типа:

- *унарные* — применимы над одним операндом;
- *бинарные* — действуют на два операнда;
- *прочие* — не относящиеся к первым двум типам операций.

Рассмотрим операции, относящиеся к вышеперечисленным типам.

Унарные операции

К унарным операциям относятся: `-` *(минус)*, `+` *(плюс)*, `++` *(инкремент)*, `--` *(декремент)*, `!` *(логическое отрицание)* и `~` *(побитовое дополнение)*. Рассмотрим их.

Операция `-` (минус) меняет знак величины операнда на противоположный.

Примеры:

```
int a = -22;
float b = 22.345;
decimal c = 2434;
int na;
float nb;
decimal nc;
na = -a;   //  na = 22
nb = -b;   //  nb = -22.345
nc = -c;   //  nc = -2434
```

Операция `+` (плюс) никаким образом не воздействует на операнд, с которым она связана. Данная операция всего лишь дополняет операцию `-` (минус).

Операция `++` (инкремент) увеличивает значение операнда на единицу. В зависимости от того, перед или после операнда стоит знак данной операции, эффект от данной операции будет различным. Рассмотрим примеры:

```
  //  знак операции перед операндом
int a = 1;
int b;
b = ++a;    //  результат b = 2, a = 2
  //  знак операции после операнда
int a = 1;
int b;
b = a++;    //  результат b = 1, a = 2
```

Операция -- (декремент) уменьшает значение операнда на единицу. По своему действию похожа на операцию инкремента. Примеры:

```
  //  знак операции перед операндом
int a = 10;
int b;
b = --a;    //  результат b = 9, a = 9
  //  знак операции после операнда
int a = 10;
int b;
b = a++;    //  результат b = 10, a = 9
```

Операция ! (логическое отрицание) применяется только над операндом типа bool. Значение операнда, равное true, будет преобразовано в false, и наоборот. Пример:

```
bool a = true;
bool b = false;
a = !a;    //  a = false;
b = !b;    //  b = true;
```

Операция ~ (побитовое дополнение) побитно инвертирует двоичное представление значения операнда. Все биты, равные 0, превращаются в 1, и наоборот. Например:

```
byte a = 23;    //  двоичное число 00010111 соответствует числу
                //  23 в десятичной форме
byte b = (byte) ~a;    //  результат — число 11101000 в двоичной или
                       //  232 в десятичной форме
```

Бинарные операции

Рассмотрим бинарные операции, которые имеются в языке C# (и C++).

Операция *сложение* (+) — используется для сложения двух величин (значений операндов). Знак сложения размещается между операндами.

Пример:

```
int a = 10;
int b = 23;
int c;
c = a + b;    //  c = 33
```

Операция *вычитание* (-) — используется для вычитания из значения первого операнда значение второго операнда. Знак вычитания размещается между операндами. Пример:

```
int a = 100;
int b = 2;
int c;
c = a – b;    //  c = 98
```

Операция *умножение* (*) — используется для получения произведения двух операндов. Пример:

```
int a = 100;
int b = 2;
int c;
c = a * b;    //  c = 200
```

Операция *деление* (/) — используется для деления первого операнда на второй операнд. Пример:

```
int a = 100;
int b = 2;
int c;
c = a / b;    //  c = 50
```

Операция *деление по модулю* (%) — используется для деления первого операнда на второй операнд и получения в качестве результата остатка от деления. Пример:

```
int a = 55;
int b = 10;
int c;
c = a % b;    //  c = 5
```

Операция *сдвиг влево* (<<) — используется для сдвига всех битов значения первого операнда влево на количество позиций, указанных во втором операнде. Младшие биты (расположенные в правой части двоичного представления числа) при этом заполняются нулями, а старшие (расположенные в левой части) теряются, если число выходит за пределы диапазона для данного типа. Операция сдвига влево может применяться над данными типов `int`, `uint`, `long` и `ulong`.

Пример:

```
uint a = 4058482644;     //  a = 11110001111001111000011111010100b
uint b;
b = a << 6;              //  b = 11110011110000111111010100000000b
```

Операция *сдвиг вправо* (>>) — используется для сдвига всех битов значения первого операнда вправо на количество позиций, указанных во втором операнде. Операция сдвига вправо может применяться над данными типов `int`, `uint`, `long` и `ulong`. Типы `uint` и `ulong`, а также положительные `int` и `long` заполняют биты слева нулями. Отрицательные значения типа `int` и `long` оставляют единицу в позиции знакового бита, а остальные позиции заполняются нулями. Естественно, что при сдвиге вправо значения сдвинутых крайних правых битов теряются. Пример:

```
uint a = 4058482644;     //  a = 11110001111001111000011111010100b
uint b;
b = a >> 6;              //  b = 00000011110001111001111000011111b
int c = -1;              //  c = 11111111111111111111111111111111b
int d;
d = c >> 8;              //  d = 10000000111111111111111111111111b
```

Операция *равенство* (==) — используется для проверки равносильности двух выражений. Данная операция возвращает значение типа `bool`. Пример:

```
int a = 10;
int b = 64;
bool c;
c = a == b;     //  c = false
```

Операция *не равно* (!=) — используется для проверки не равенства двух выражений. Данная операция возвращает значение типа `bool`. Пример:

```
int a = 10;
int b = 64;
bool c;
c = a != b;     //  c = true
```

Операция *меньше* (<) — используется для проверки, является ли значение первого выражения меньше значения второго выражения. Данная операция возвращает значение типа `bool`. Пример:

```
int a = 10;
int b = 64;
bool c;
c = a < b;     //  c = true
```

Операция *меньше либо равно* (<=) — используется для проверки, является ли значение первого выражения меньше либо равным значению второго выражения. Данная операция возвращает значение типа `bool`. Пример:

```
int a = 10;
int b = 10;
bool c;
c = a <= b;    //  c = true
```

Операция *больше* (>) — используется для проверки, является ли значение первого выражения больше значения второго выражения. Данная операция возвращает значение типа `bool`. Пример:

```
int a = 10;
int b = 64;
bool c;
c = a > b;    //  c = false
```

Операция *больше либо равно* (>=) — используется для проверки, является ли значение первого выражения больше либо равным значению второго выражения. Данная операция возвращает значение типа `bool`. Пример:

```
int a = 10;
int b = 64;
bool c;
c = a >= b;    //  c = false
```

Над данными типов `int`, `uint`, `long` и `ulong` можно производить поразрядные операции, приведенные в табл. 11.7. Особенностью этих операций является то, что они выполняются побитно над операндами, переведенными в двоичную форму. Результат выдается в десятичной системе счисления.

Таблица 11.7. *Поразрядные арифметические операции*

Знак операции	Операция	Пример
`&`	Поразрядное умножение	`X & Y`
`|`	Поразрядное сложение	`X | Y`
`^`	Поразрядное исключающее ИЛИ	`X ^ Y`

В табл. 11.8 приведены результаты выполнения поразрядных арифметических операций (табл. 11.7).

Операции производятся поразрядно справа налево, затем результат переводится в десятичную систему счисления.

Таблица 11.8. *Результаты выполнения поразрядных арифметических операций*

Знак операции	Операция	Бит 1	Бит 2	Результирующий бит
&	Поразрядное умножение	0	0	0
		0	1	0
		1	0	0
		1	1	1
|	Поразрядное сложение	0	0	0
		0	1	1
		1	0	1
		1	1	1
^	Поразрядное исключающее ИЛИ	0	0	0
		0	1	1
		1	0	1
		1	1	0

Приведу примеры использования побитовых арифметических операций (листинг 11.8).

Листинг 11.8. Использование побитовых операций

```
  //  объявление трех целочисленных переменных
int a = 175;    //  a = 10101111b
int b = 77;     //  b = 1001101b
int c;
  //  выполнение операций
c = a | b;      //  результат — число 239
c = a & b;      //  результат — число 13
c = a ^ b;      //  результат — число 226
```

Логические операции над двумя операндами, доступные в языках C++ и C#, приведены в табл. 11.9.

В табл. 11.10 приведены результаты выполнения логических операций.

Таблица 11.9. *Логические операции*

Знак операции	Операция	Типы операндов	Тип результата	Пример
`&`	Конъюнкция (логическое И)	`bool`	`bool`	`A & B`
`|`	Дизъюнкция (логическое ИЛИ)	`bool`	`bool`	`A | B`
`^`	Исключающая дизъюнкция (исключающее ИЛИ)	`bool`	`bool`	`A ^ B`
`&&`	Условная операция И	`bool`	`bool`	`A && B`
`||`	Условная операция ИЛИ	`bool`	`bool`	`A || B`

Таблица 11.10. *Результаты выполнения логических операций*

Знак операции	Операция	Операнд 1	Операнд 2	Результат
`&`	Конъюнкция (логическое И)	`false`	`false`	`false`
		`false`	`true`	`false`
		`true`	`false`	`false`
		`true`	`true`	`true`
`|`	Дизъюнкция (логическое ИЛИ)	`false`	`false`	`false`
		`false`	`true`	`true`
		`true`	`false`	`true`
		`true`	`true`	`true`
`^`	Исключающая дизъюнкция (исключающее ИЛИ)	`false`	`false`	`false`
		`false`	`true`	`true`
		`true`	`false`	`true`
		`true`	`true`	`false`

Условные операции И и ИЛИ похожи на операции И и ИЛИ соответственно. Но оценка значения выражения происходит по первому операнду, без сопоставления второго операнда. Таким образом, если первый операнд при выполнении условной операции И имеет значение `false`, то значение всего выражения будет `false`. Если первый операнд при выполнении условной операции ИЛИ имеет значение `true`, то и значение всего выражения будет `true`.

Последняя серия бинарных операций — это операции *присваивания*. Самая простая ее форма уже применялась неоднократно. Это выражение вида:

```
a = b;
```

Знак простой операции присваивания — это знак равенства (=).

Кроме простой операции присваивания в своих программах на языках C++ и C#, вы можете использовать составные операции присваивания, которые состоят из любой арифметической операции и простой операции присваивания, например:

```
int a = 10;
a -=6;     //  a = 4
```

Аналог данного примера может выглядеть так:

```
int a = 10;
a = a – 6; // a = 4
```

В табл. 11.11 представлен список допустимых составных операций присваивания.

Таблица 11.11. *Составные операции присваивания*

Операция	Назначение	Операция	Назначение
+=	Сложение	<<=	Сдвиг влево
-=	Вычитание	>>=	Сдвиг вправо
*=	Умножение	&=	Поразрядное И
/=	Деление	|=	Поразрядное ИЛИ
%	Остаток от деления	^=	Поразрядное исключающее ИЛИ

Прочие операции

Имеется несколько операций, которые не могут быть отнесены к тому или иному типу операций. Это такие операции, как `is`, `as`, `sizeof()`, `typeof()`, `checked()` и `unchecked()`. Рассмотрим их.

Операция *is*

Данная операция проверяет переменную на предмет ее принадлежности к определенному типу. Если переменная принадлежит этому типу, то возвращается значение `true`, иначе — `false`.

Пример:

```
int a = 10;
bool b = i is int;     // b = true
bool c = i is decimal; // c = false
```

Операция *as*

Эта операция позволяет преобразовать один тип в другой. В примере значения различных объектов преобразовываются в тип `string`.

```
using System;
class MyClass1
{
}

class MyClass2
{
}

public class IsTest
{
   public static void Main()
   {
     object [] myObjects = new object[6];
     myObjects[0] = new MyClass1();
     myObjects[1] = new MyClass2();
     myObjects[2] = "hello";
     myObjects[3] = 123;
     myObjects[4] = 123.4;
     myObjects[5] = null;

     for (int i = 0; i < myObjects.Length; ++i)
     {
       string s = myObjects[i] as string;
       Console.Write ("{0}:", i);
       if (s != null)
       Console.WriteLine ( "'" + s + "'");
       else
       Console.WriteLine ( " не является строкой");
     }
   }
}
```

В результате выполнения примера на экран будет выведена следующая информация:

```
0: не является строкой
1: не является строкой
2:'hello'
3: не является строкой
4: не является строкой
5: не является строкой
```

Операция *sizeof()*

Язык C# позволяет использовать низкоуровневые функции с помощью конструкции, известной как код *unsafe*. Операция `sizeof()` применима только внутри кода unsafe. Она принимает в качестве параметра тип и возвращает размер его объекта в байтах. Рассмотрим пример:

```
using System;
class SizeClass
{
   unsafe public static void SizesOf()
   {
      Console.WriteLine("Размер типа short — {0}.", sizeof(short));
      Console.WriteLine("Размер типа int — {0}.", sizeof(int));
      Console.WriteLine("Размер типа long — {0}.", sizeof(long));
   }
}

class MainClass
{
   public static void Main()
   {
      SizeClass.SizesOf();
   }
}
```

В результате выполнения данного примера на экране появится такая информация:

```
Размер типа short — 2.
Размер типа int — 4.
Размер типа long — 8.
```

Операция *typeof()*

Данная операция возвращает объект класса `Type`. Класс `Type` содержит информацию о размере или типе ссылки для данного типа. Операция `typeof()` используется для раскрытия информации о типах ссылки и значения.

Рассмотрим пример:

```
using System;
using System.Reflection;

public class MyClass
{
   public int intI;
   public void MyMeth()
   {
   }

   public static void Main()
   {
      Type t = typeof(MyClass);

      MethodInfo[] x = t.GetMethods();
      foreach (MethodInfo xtemp in x)
      {
         Console.WriteLine(xtemp.ToString());
      }

      Console.WriteLine();

      MemberInfo[] x2 = t.GetMembers();
      foreach (MemberInfo xtemp2 in x2)
      {
         Console.WriteLine(xtemp2.ToString());
      }
   }
}
```

В результате выполнения данного примера на экране появится следующая информация:

```
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void MyMeth()
Void Main()
System.Type GetType()

Int32 intI
Int32 GetHashCode()
Boolean Equals(System.Object)
```

```
System.String ToString()
Void MyMeth()
Void Main()
System.Type GetType()
Void .ctor()
```

Операция *checked()*

Эта операция реагирует на возникновение переполнения для некоторых операций. Рассмотрим пример:

```
  //  переполнение переменных выражений проверяется во время выполнения
  //  приложения
using System;

class OverFlowTest
{
   static short x = 32767;    //  максимальное значение для типа short
   static short y = 32767;

   //  используется оператор checked
   public static int myMethodCh()
   {
      int z = 0;
      try
      {
         z = checked((short)(x + y));
      }
      catch (System.OverflowException e)
      {
      System.Console.WriteLine(e.ToString());
      }
      return z;               // генерирует исключение OverflowException
   }

   public static void Main()
   {
       Console.WriteLine("Выходное значение: {0}", myMethodCh());
   }
}
```

В результате выполнения данного примера на экран будет выведена информация, представленная на рис. 11.3.

Рис. 11.3. Пример работы операции `checked()`

Операция *unchecked()*

В случае когда нужно проигнорировать ошибку и получить результат вне зависимости от возникновения переполнения, вы можете использовать операцию `unchecked()`.

Рассмотрим пример:

```
using System;

class OverFlowTest
{
   static short x = 32767;     //  максимальное значение для типа short
   static short y = 32767;

   public static int myMethodUnch()
   {
      int z = unchecked((short)(x + y));
      return z;                //  возвращает значение -2
   }

   public static void Main()
   {
      Console.WriteLine("Значение: {0}", myMethodUnch());
   }
}
```

В результате выполнения программы на экране появится следующая запись:

```
Значение: -2
```

Порядок выполнения операций

В языках C++ и C# существуют правила, которые обеспечивают достоверность результата при вычислении выражения. Эти правила регулируются *старшинством* и *ассоциативностью* операций.

Старшинство связано с порядком, в котором должны выполняться операторы.

Ассоциативность бывает левая и правая. Левая подразумевает, что операции должны выполняться слева направо, а правая — справа налево.

В табл. 11.12 приведены операции языка C#, а также их старшинство (чем выше в таблице расположен оператор, тем он старше) и ассоциативность.

Таблица 11.12. *Старшинство и ассоциативность операций C#*

Операции	Ассоциативность
`(x)`, `x.y`, `f(x)`, `a[x]`, `x++`, `x--`, `new`, `typeof()`, `sizeof()`, `checked()`, `unchecked`	Левая
Унарный `+`, унарный `-`, `~`, `++x`, `--X`, `(T)x`	Левая
`*`, `/`, `%`	Левая
Арифметический `+`, арифметический `-`	Левая
`<<`, `>>`	Левая
`<`, `>`, `<=`, `>=`, `is`, `as`	Левая
`==`, `!=`	Левая
`&`	Левая
`^`	Левая
`|`	Левая
`&&`	Левая
`||`	Левая
`?:`	Правая
`=`, `*=`, `/=`, `%=`, `+=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	Правая

Операторы

Операторы — это команды компилятору языка на выполнение определенных действий.

Все операторы можно условно разделить на *простые* и *структурированные*.

Простые операторы не изменяют хода выполнения программы. К таким операторам относятся простые операции, уже рассмотренные нами.

Структурированные операторы — это операторы, которые изменяют ход выполнения команд программы.

К числу структурированных операторов можно отнести:

- оператор *перехода*;
- оператор *условия*;
- оператор *выбора*;
- операторы *цикла*.

Рассмотрим эти операторы подробнее.

Оператор перехода

Оператор перехода позволяет перейти от текущего места выполнения программы в другое место, которое не является следующим по порядку. Данный оператор нарушает нормальный ход выполнения программы.

Переход осуществляется при помощи так называемых *меток*.

Метка — это идентификатор, который находится перед каким-либо оператором программы и отделен от него двоеточием.

Оператор перехода выглядит так:

```
goto метка;
```

В языке C++ операторы `goto` могут передавать управление любому месту программы. В языке C# операторы `goto` должны всегда осуществлять переход на тот же самый уровень, либо на более высокий, выходя из программного блока.

Рассмотрим фрагмент программы, в которой используется метка и оператор `goto`:

```
int a = 10;
goto l1;   //  переход на метку l1
a += 10;   //  этот оператор не будет выполняться никогда
l1: a *=5; //  оператор, помеченный меткой l1
```

Примечание

Использование меток и операторов перехода *не является хорошим тоном программирования*. Старайтесь по возможности не применять этот оператор, т. к. его использование затрудняет чтение текста программы.

Забегая немного вперед, отмечу, что оператор `goto` не может вызвать переход внутрь цикла.

Оператор условия

Оператор условия предназначен для выполнения или не выполнения каких-либо действий, зависящих от результата условия типа `bool`.

Примечание

Многие операторы, которые будут рассматриваться далее в этой главе, содержат логические выражения, которые дают возможность выбора вариантов. Программы на языке C++ могут рассматривать положительные целые значения, как значения типа `true`. В языке C# этого нет. Логическое выражение в языке C# должно возвращать значение `true` или `false`.

Этот оператор применяют для разветвления выполнения программы. Если данное условие истинно (и только в этом случае!), то выполняется некоторая последовательность операторов, иначе — выполняются другие операторы.

Язык C# предоставляет программисту три варианта оператора условия `if`:

- простой `if`;
- конструкция `if...then...else`;
- конструкция `if...else if...else`.

Простой *if*

Вид оператора простого `if` следующий:

```
if (логическое выражение)
[{]
   операторы, выполняемые если логическое выражение истинно
[}]
```

Рассмотрим пример использования оператора `if`:

```
if (args.Lifength == 1)
{
   Console.WriteLine("Проба, {0}", args[0]);
}
```

Примечание

Использование фигурных скобок необязательно в том случае, когда нужно выполнить только одно действие в случае истинности логического выражения. Но использование скобок всегда является хорошим тоном программирования и позволяет избежать непредсказуемых ошибок.

Оператор *if...then...else*

Для поддержки обоих условий — `true` и `false` — используется оператор `if...then...else`. Этот оператор имеет вид:

```
if (логическое выражение)
[{]
   операторы, выполняемые если логическое выражение истинно
[}]
else
[{]
   операторы, выполняемые если логическое выражение ложно
[}]
```

Пример:

```
if (args.Length == 0)
{
   Console.WriteLine("Вы ничего не ввели в качестве аргумента.");
}
else
{
   Console.WriteLine("Вы ввели в качестве аргумента: {0}", args[0]);
}
```

Оператор *if...else if...else*

Иногда возникает необходимость оценить несколько условий для выбора необходимого действия. В этом случае используется оператор `if...else if...else`. Его вид следующий:

```
if (логическое выражение)
[{]
   операторы, выполняемые если логическое выражение истинно
[}]
else if (логическое выражение)
[{]
   операторы, выполняемые если логическое выражение истинно
[}]
...
else if (логическое выражение)
[{]
   операторы, выполняемые если логическое выражение истинно
[}]
else
[{]
   операторы, выполняемые если логическое выражение ложно
[}]
```

Рассмотрим пример использования данного оператора:

```
if (args.Length == 0)
{
   Console.WriteLine("Вы ничего не ввели в качестве аргумента.");
}
else if (args.Length == 1)
{
   Console.WriteLine("Вы ввели в качестве аргумента: {0}", args[0]);
}
else
{
   Console.WriteLine("Вы ввели несколько аргументов");
}
```

Оператор выбора

Оператор выбора (`switch`) — это конструкция языка, позволяющая сделать выбор из произвольного числа вариантов и выполнить в соответствии с этим выбором определенные действия. То есть, оператор выбора позволяет обойтись без использования оператора `if...else if...else`.

Примечание

Использование оператора выбора вместо оператора условия `if...else if...else` является хорошим тоном программирования.

Оператор выбора имеет следующий вид:

```
switch (целочисленное или строковое значение)
{
   case литерал_1:
   операторы
   break;
...
   case литерал_N:
   операторы
   break;
   [default:
   операторы]
}
```

Оператор `switch` в языке C++ может принимать только целочисленные значения. Язык C# может кроме целочисленных значений принимать строки и перечисления.

Возможность "проваливаться" сквозь конструкцию `case`, которая есть в языке C++, отсутствует в языке C#. В языке C# операторы `break` являются обязательными. Исключение составляют только пустые конструкции `case` (не имеющие операторов).

Рассмотрим пример использования оператора `switch`:

```
char a;
...
switch (a)
{
   case "А":
   Console.WriteLine("Буква А");
   break;
   case "Б":
   Console.WriteLine("Буква Б");
   break;
   case "В":
   Console.WriteLine("Буква В");
   break;
   default:
   Console.WriteLine("Неизвестная буква");
   break;
}
```

Как уже было замечено, оператор `break` является обязательным, но могут быть и исключения. Например:

```
char a;
...
switch (a)
{
   case "а":
   case "А":
   Console.WriteLine("Буква А");
   break;
   case "б":
   case "Б":
   Console.WriteLine("Буква Б");
   break;
   case "в":
   case "В":
   Console.WriteLine("Буква В");
   break;
```

```
    default:
    Console.WriteLine("Неизвестная буква");
    break;
}
```

Примечание

Несмотря на тот факт, что наличие `default` в операторе `switch` не обязательно, его стоит включать во избежание ошибок, которые могут возникнуть, если ни один из вариантов `case` не оказался подходящим.

Операторы цикла

Иногда необходимо, чтобы какая-либо часть программы выполнялась несколько раз. Для этого во всех языках программирования используются *циклы*.

Цикл — это последовательность операторов, команд, которая выполняется более одного раза, т. е. повторяется. Эти повторяющиеся операторы и команды называют *телом цикла*.

В языке C# имеется четыре вида операторов цикла:

- цикл `while`;
- цикл `do`;
- цикл `for`;
- цикл `foreach` (данный вид цикла отсутствует в языке C++).

Каждый из этих циклов удобно применять в различных случаях, хотя практически всегда, во всей программе, можно обойтись использованием лишь одного из этих циклов.

В тело цикла допускается помещать оператор `break`. Он приводит к прекращению выполнения операторов тела цикла и осуществляет переход к выполнению операторов, расположенных сразу за циклом.

Вы можете досрочно завершить выполнение данного повторения цикла при помощи оператора `continue`. Этот оператор передает управление сразу в конец цикла, что досрочно завершает данное повторение.

Операторы циклов допускают множественные вложения друг в друга. При вложении сначала выполняются внутренние операторы цикла, а затем внешние. Далее приведу примеры вложений операторов цикла друг в друга.

Цикл *while*

Цикл `while` применяется в случае, когда число повторений заранее неизвестно и если при некоторых условиях тело цикла может не выполняться совсем. Данный цикл выглядит следующим образом:

```
while (условие)
[{]
 операторы, исполняемые если условие истинно
[}]
```

`условие` представляет собой логическое выражение, которое может быть ложным или истинным.

Данный цикл выполняется в том случае, когда условие имеет значение `true`. Как только условие примет значение `false` — цикл выполняться не будет. Данный вид цикла не выполняется совсем, если при первоначальной проверке условия оно имеет значение `false`.

Рассмотрим пример оператора цикла `while`:

```
int i = 10;
int s = 1;
  //  начало цикла
while (i > 0)
{
  //  тело цикла
  s = s + i;
  i = i – 1;    //  изменили переменную i самостоятельно
}
  //  конец цикла
```

Примечание

В отличие от цикла `for`, при использовании других видов цикла следите за тем, чтобы условие цикла когда-нибудь изменилось. Для этого нужно изменять переменные, входящие в условие цикла тела цикла. В противном случае может возникнуть ситуация, когда цикл будет выполняться бесконечно.

Пример бесконечного цикла такой:

```
int i = 1;
int s = 1;
while (i == 1)
{
   s = s + i;
}
```

Обратите внимание на возможную ошибку, которая может возникнуть при использовании циклов, — ошибку *пустого оператора*. Пример:

```
int i = 10;
int s = 1;
```

```
while (i > 0);    //  пустой оператор
{
   //  эти операторы не будут выполнены никогда
   s = s + i;
   i = i – 1;
}
```

Так как фигурные скобки не являются обязательными, точка с запятой после условия расценивается как оператор цикла, и именно он будет выполняться бесконечное число раз.

Цикл *do*

Цикл `do` используется в случае, когда желательно, чтобы тело цикла выполнилось хотя бы один раз, а общее количество повторений цикла заранее неизвестно.

Данный цикл имеет следующий вид:

```
do
{
   оператор1;
   ...
   операторN;
}
while (условие);
```

Операторы, расположенные внутри фигурных скобок, представляют собой тело цикла. `условие` — это логическое выражение.

Операторы цикла выполняются, по крайней мере, один раз, а затем происходит проверка истинности условия. Если условие является ложным (`false`), то операторы цикла выполняются повторно. Цикл будет выполняться до тех пор, пока условие цикла не станет истинным (`true`).

Рассмотрим пример использования цикла `do`:

```
  //  начало цикла
do
{
   //  тело цикла
   k = i + j;
   i = j;
   j = k;
}
while j = 0;    //  условие цикла
```

Цикл *for*

Цикл `for` применяется в том случае, когда заранее известно, сколько раз он должен выполниться. Данный цикл выглядит следующим образом:

```
for (инициализатор; логическое выражение; модификатор)
[{]
   операторы;
[}]
```

Инициализатор цикла выполняется всего один раз в начале цикла. После выполнения инициализатора оценивается логическое выражение. Чтобы осуществлялись операторы тела цикла, логическое выражение должно иметь значение `true`. После исполнения операторов тела цикла выполняется модификатор, затем логическое выражение оценивается снова. Весь этот процесс продолжается до тех пор, пока значение логического выражения не станет равным `false`. Тогда выполнение цикла прекращается.

В соответствии со стандартом C++, объявление инициализатора цикла `for` определяет в необходимом блоке новую область видимости для переменной с тем же самым именем. В языке C# не допускается, чтобы переменная в инициализаторе цикла `for` перекрывала переменную с тем же самым именем в соответствующем блоке, поэтому в данном случае возникнет ошибка.

Рассмотрим примеры с использованием цикла `for`:

```
int i, j, a, s;
  //  начало цикла
for (i = 1; i == 20; i++)
{
   //  тело цикла
   s = s + i;
   a = a * i;
}
  //  конец цикла, значение i увеличивается на 1
  //  начало внешнего цикла
for (i = 1; i == 10; i++)
{
  //  тело цикла
  //  начало внутреннего цикла
   for (j = 10; j == 1; j--)
   {
      // тело внутреннего цикла
      s = s + j;
      a = s + a;
   }
   //  конец внутреннего цикла, i уменьшается на 1
```

```
   a = a * s;
}
   //  конец внешнего цикла, i увеличивается на 1
```

Цикл *foreach*

Цикл `foreach` применяется для итеративных действий над коллекциями. Синтаксис данного цикла следующий:

```
foreach (тип идентификатор in коллекция)
[{]
   операторы;
[}]
```

Типом может быть любой тип языка C# (напомним, что в C++ цикла `foreach` нет) или тип, определенный пользователем. *Идентификатор* представляет собой имя переменной, которая должна быть использована в цикле. *Коллекция* — это любой объект коллекции.

При выполнении цикла последовательно перебираются все элементы коллекции с первого до последнего. Рассмотрим пример с использованием коллекций:

```
using System;

  //  объявление коллекции
public class MyCollection
{
   int[] items;

   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   public class MyEnumerator
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
```

```
      {
         collection = coll;
         nIndex = -1;
      }

      public bool MoveNext()
      {
         nIndex++;
         return(nIndex < collection.items.GetLength(0));
      }

      public int Current
      {
         get
         {
            return(collection.items[nIndex]);
         }
      }
      }
   }

public class MainClass
{
   public static void Main()
   {
      MyCollection col = new MyCollection();
      Console.WriteLine("Значения в коллекции:");

      //  отображение содержимого коллекции
      foreach (int i in col)
      {
         Console.WriteLine(i);
      }
   }
}
```

В результате выполнения данного примера на экране появится следующая информация:

```
Значения в коллекции:
12
44
33
2
50
```

Так как массив (`array`) является встроенной коллекцией языка C#, то цикл `foreach` можно использовать и при работе с элементами массивов. Рассмотрим пример, находящий четные и нечетные числа массивов:

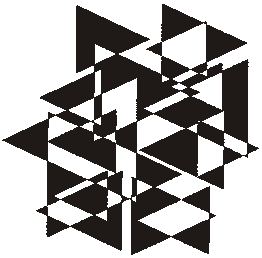
```
using System;
class MainClass
{
   public static void Main()
   {
      int odd = 0, even = 0;
      int[] arr = new int [] { 0, 1, 2, 5, 7, 8, 11 };

      foreach (int i in arr)
      {
         if (i%2 == 0)
         even++;
         else
         odd++;
      }

   Console.WriteLine("Найдено {0} нечетных чисел, и {1} четных чисел.",
   odd, even);
   }
}
```

В результате работы данного примера на экране появится следующее сообщение:

```
Найдено 4 нечетных чисел, и 3 четных чисел.
```

Итак, мы рассмотрели основы синтаксиса языка C# и главные отличия данного языка от C++. В *гл. 12* продолжим знакомство с возможностями языка C#.

ГЛАВА 12



Объектное и компонентное программирование на C++ и C#

В данной главе рассматривается работа с объектами и классами. Изучаются понятия делегатов и событий, структур и интерфейсов. Кроме того, затрагиваются вопросы обработки исключительных ситуаций. Как и ранее, будет приведено много примеров на C#, и пойдет речь об отличиях в C++, если таковые имеются.

Объекты и компоненты

Объекты в C# определяются с помощью типа `class` (класс). *Атрибуты класса* представляют собой *поля*. Для реализации поведения объекта используются *методы*.

Рассмотрим пример описания объекта:

```
class MyClass
{
   int a;
   int b;
   bool c;

   void Menshe()
   {
      //  реализация поведения
   }
}
```

Вы можете создавать также и *абстрактные классы*. Для предотвращения возможности создания экземпляров абстрактного класса перед ключевым словом `class` ставится слово `abstract`. Например:

```
abstract class Car
{
   //  абстрактное определение и реализация класса
}
class Volvo : Car
{
   //  реализация класса
}
```

В этом примере класс `Volvo` является наследником класса `Car`, что определяется установкой двоеточия после названия класса `Volvo`.

Языки C# и C++ поддерживают основные понятия объектно-ориентированного программирования: *инкапсуляцию*, *наследование* и *полиморфизм*. Наследование мы уже рассмотрели. Теперь перейдем к инкапсуляции.

Понятие *инкапсуляции* применяется вместо выражения "объект внутри объекта". Рассмотрим пример реализации инкапсуляции в языке C#:

```
class Engine
{
   //  определение и реализация класса
}
class Car
{
   Engine VolvoEngine;
   //  определение и реализация класса
}
```

В этом примере определяется два класса — `Engine` и `Car`. Внутри класса `Car` объявлен объект `Engine` с именем `VolvoEngine`. Таким образом, установлено отношение "автомобиль имеет двигатель". Атрибутами и поведением объекта `VolvoEngine` управляет класс `Engine`.

Полиморфизм — это возможность методов различных объектов иметь одинаковые имена, но отличаться по своему содержанию. Это достигается в результате переопределения метода объекта-предка в объекте-наследнике. При этом обращение к одному и тому же методу у объекта-предка и объекта-потомка может привести к разным результатам.

Рассмотрим пример реализации полиморфизма:

```
using System;
class MainClass
{
   public virtual void SayText()
```

```
   {
      Console.WriteLine("Этот текст не будет выведен на консоль");
   }
}
class My1 : MainClass
{
   public override void SayText()
   {
      Console.WriteLine("Первый текст");
   }
}
class My2 : MainClass
{
   public override void SayText()
   {
      Console.WriteLine("Второй текст");
   }
}
class My3 : MainClass
{
   public override void SayText()
   {
      Console.WriteLine("Третий текст");
   }
}
class Go
{
   public static void Main()
   {
      MainClass[] MC = new MainClass[3];
      MC[0] = new My1();
      MC[1] = new My2();
      MC[2] = new My3();
      foreach (MainClass a in MC)
      {
         a.SayText();
      }
   }
}
```

В результате работы данного примера на экране появится следующее:

```
Первый текст
Второй текст
Третий текст
```

Интерфейсы

Классы в языке C# претерпели довольно серьезные изменения по сравнению с языком программирования C++. Первое, что бросается в глаза, — это невозможность множественного наследования. Такой подход уже знаком тем, кто пишет на языках Object Pascal и Java, а вот программисты C++ могут быть несколько озадачены. Хотя при более близком рассмотрении данное ограничение уже не кажется серьезным или непродуманным. Во-первых, множественное наследование, реализованное в C++, нередко являлось причиной нетривиальных ошибок (при том, что не так уж часто приходится описывать классы с помощью множественного наследования). Во-вторых, в C#, как и в диалекте Object Pascal фирмы Borland, разрешено наследование от нескольких интерфейсов.

Интерфейсом в C# является тип ссылок, содержащий только абстрактные элементы, не имеющие реализации. Непосредственно реализация этих элементов должна содержаться в классе, производном от данного интерфейса (вы не можете напрямую создавать экземпляры интерфейсов).

Интерфейсы C# могут содержать методы, свойства и индексаторы, но в отличие, например, от Java, они не могут содержать константных значений. Рассмотрим простейший пример использования интерфейсов:

```
using System;
class CShape
{
   bool IsShape()
   {return true;}
}
interface IShape
{
   double Square();
}
class CRectangle: CShape, IShape
{
   double width;
   double height;
   public CRectangle(double width, double height)
   {
      this.width = width;
      this.height = height;
   }
   public double Square()
   {
      return (width * height);
   }
}
```

```
class CCircle: CShape, IShape
{
   double radius;
   public CCircle(double radius)
   {
      this.radius = radius;
   }
   public double Square()
   {
      return (Math.PI * radius * radius);
   }
}
class Example_3
{
   public static void Main()
   {
      CRectangle rect = new CRectangle(3, 4);
      CCircle circ = new CCircle(5);
      Console.WriteLine(rect.Square());
      Console.WriteLine(circ.Square());
   }
}
```

Оба объекта, `rect` и `circ`, являются производными от базового класса `CShape`, и тем самым они наследуют единственный метод `IsShape()`. Задав имя интерфейса `IShape` в объявлениях `CRectangle` и `CCircle`, идет указание на то, что в данных классах содержится реализация всех методов интерфейса `IShape`. Кроме того, члены интерфейсов не имеют модификаторов доступа. Их область видимости определяется непосредственно реализующим классом.

Свойства

Рассматривая классы языка C#, просто нельзя обойти такое "новшество", как *свойства* (properties). Надо сказать, что здесь чувствуется влияние языков Object Pascal и Java, в которых свойства всегда являлись неотъемлемой частью классов. Что же представляют собой эти самые свойства? С точки зрения пользователя, они выглядят практически так же, как и обычные поля класса. Им можно присваивать некоторые значения и получать их обратно. В то же время свойства имеют большую функциональность, т. к. чтение и изменение их значений выполняется с помощью специальных методов класса. Такой подход позволяет изолировать пользовательскую модель класса от ее реализации. Поясню данное определение на конкретном примере:

```
using System;
using System.Runtime.InteropServices;
class Screen
{
   [DllImport("kernel32.dll")]
   static extern bool SetConsoleTextAttribute(int hConsoleOutput, ushort
   wAttributes);
   [DllImport("kernel32.dll")]
   static extern int GetStdHandle(uint nStdHandle);
   const uint STD_OUTPUT_HANDLE = 0x0FFFFFFF5;
   static Screen()
   {
      output_handle = GetStdHandle(STD_OUTPUT_HANDLE);
      m_attributes = 7;
   }
   public static void PrintString(string str)
   {
   Console.Write(str);
   }
      public static ushort Attributes
   {
      get
      {
         return m_attributes;
      }
      set
      {
         m_attributes = value;
         SetConsoleTextAttribute(output_handle, value);
      }
   }
   private static ushort m_attributes;
   private static int  output_handle;
}
class Example_4
{
   public static void Main()
   {
      for (ushort i = 1; i < 8; i++)
      {
         Screen.Attributes = i;
         Screen.PrintString("Property Demo\n");
      }
   }
}
```

Программа выводит сообщение `Property Demo`, используя различные цвета символов.

Давайте попробуем разобраться в том, как она работает. Итак, сначала импортируются важные функции API-интерфейса Windows:

- `SetConsoleTextAttribute`;
- `GetStdHandle`.

К сожалению, стандартный класс среды .NET под названием `Console` не имеет средств управления цветом вывода текстовой информации. Пока же для этих целей придется воспользоваться службой вызова платформы `PInvoke` (обратите внимание на использование атрибута `DllImport`). Далее, в конструкторе класса `Screen` образуется стандартный дескриптор потока вывода консольного приложения, и его значение помещается в закрытую переменную `output_handle` для дальнейшего использования функцией `SetConsoleTextAttribute`. Кроме этого, переменной `m_attributes` присваивается начальное значение атрибутов экрана (7 соответствует белому цвету символов на черном фоне). Замечу, что в реальных условиях стоило бы получить текущие атрибуты экрана с помощью функции `GetConsoleScreenBufferInfo` из набора API-интерфейса Windows. В данном случае это несколько усложнило бы пример.

В классе `Screen` объявлено свойство `Attributes`, для которого определена функция чтения (`getter`) и функция записи (`setter`). Функция чтения не выполняет каких-либо специфических действий и просто возвращает значение поля `m_attributes` (в реальной программе она должна возвращать значение атрибутов, полученное с помощью все той же `GetConsoleScreenBufferInfo`). Функция записи несколько сложнее, т. к. кроме тривиального обновления значения `m_attributes` она вызывает функцию `SetConsoleTextAttribute`, устанавливая заданные атрибуты функций вывода текста. Значение устанавливаемых атрибутов передается специальной переменной `value`. Обратите внимание на то, что поле `m_attributes` является закрытым, а стало быть, оно не может быть доступно вне класса `Screen`. Единственным способом чтения и изменения этого метода является свойство `Attributes`.

Свойства позволяют не только возвращать и изменять значение внутренней переменной класса, но и выполнять дополнительные функции. Так, они позволяют произвести проверку значения или выполнить иные действия, как показано в приведенном примере.

В языке C# свойства реализованы на уровне синтаксиса. Более того, рекомендуется вообще не использовать открытых полей класса. На первый взгляд, при таком подходе теряется эффективность из-за того, что операции присваивания будут заменены вызовами функций `getter` и `setter`, но это не так. Среда .NET сгенерирует для них соответствующий код.

Использование делегатов и событий

Неотъемлемой частью в практике программирования стало использование *обратных вызовов* (callback) и *уведомлений* (notifications). Основанные на них приемы нашли широкое применение при создании кода. К тому же, в последнее время все больше распространяется программирование с использованием событий (замечу, что события реализуются с помощью уведомлений).

Для реализации обратных вызовов и уведомлений в языках C и C++ используют указатели на функции. Такой подход несет в себе некоторые недостатки, т. к. не предоставляет типовой защищенности, что может привести к появлению ошибок, связанных с преобразованием типов, которые не выявляются на стадии компиляции.

Поскольку типовая защищенность — одно из достоинств C#, в этом языке вы не можете объявить указатель на функцию. Для реализации вышеупомянутых действий (обратных вызовов и уведомлений) в C# используются *делегаты*.

Делегат представляет собой подобие указателя на функцию. Он сохраняет преимущества указателя на функцию, при этом не имеет его (указателя) недостатков.

Использование делегатов позволяет добиться тех же результатов, что и при использовании указателей на функции в C/C++, не теряя типовой защищенности.

Объявление делегатов

Поскольку CLR является объектно-ориентированной средой, в ней все связано с классами и объектами. Это относится и к делегатам.

Делегаты в CLR реализованы в виде классов, производных от базового класса делегата:

```
System.Delegate
```

или

```
System.MulticastDelegate
```

Строго говоря, *делегат* — это класс, содержащий данные о сигнатуре метода.

Экземпляр делегата — это объект, который позволяет привязаться к конкретному методу, соответствующему определенной сигнатуре (подобно объявлению указателя на функцию в C/C++).

Как и указатель на функцию, делегат может быть передан в виде аргумента.

Рассмотрим действия, которые надо проделать, чтобы воспользоваться делегатами. Для начала необходимо объявить делегат. Для объявления делегатов в C# используется ключевое слово `delegate`. Пример объявления делегата может быть таким:

```
delegate string MyDelegate(int a);
```

Все, что мы получили от данного объявления, — это новый тип (класс). Данный класс определяет форму метода — его сигнатуру. В сигнатуру метода входит тип возвращаемого значения и список аргументов. В данном случае, метод должен возвращать строку и принимать один целочисленный аргумент. Так как делегат является классом, он может быть объявлен как в пространстве имен, так и в классе. В листингах 12.1 и 12.2 содержатся корректные объявления делегатов.

Листинг 12.1. Объявление делегата № 1

```
using System;
delegate string MyDelegate(int a);
class MyApp
{
   public static void Main()
   {
      ...
   }
}
```

Листинг 12.2. Объявление делегата № 2

```
using System;
class MyApp
{
   delegate string MyDelegate(int a);
   public static void Main()
   {
      ...
   }
}
```

Во втором примере определяется вложенный класс.

В определении делегата (перед ключевым словом `delegate`) могут быть указаны модификаторы доступа, их смысл идентичен смыслу модификаторов доступа обычного класса.

Итак, делегат определен. Теперь необходимо создать его экземпляр. Экземпляр делегата является объектом и создается подобно экземпляру любого класса:

```
MyDelegate dg1;  // объявление экземпляра делегата
```

Здесь `MyDelegate` — имя класса, а `dg1` — имя объекта. Перед использованием делегата его необходимо инициализировать. Экземпляр делегата инициализируется, как и экземпляр любого класса, при помощи оператора `new`:

```
MyDelegate dg1;   //  объявление экземпляра делегата
dg1 = new MyDelegate(MyMethod);    //  инициализация экземпляра делегата
```

Как видно из фрагмента программы, конструктору делегата передается идентификатор метода, на который будет указывать экземпляр делегата. Отличие заключается в том, что при попытке передать идентификатор метода конструктору обычного класса, независимо от типа аргумента конструктора, будет выдано сообщение об ошибке. Данный факт объясняется тем, что концепция делегатов поддерживается синтаксисом C#. Передача имени метода допустима только конструкторам делегатов. Как и для обычных классов, при объявлении экземпляр делегата можно инициализировать:

```
// объявление и инициализация экземпляра делегата
MyDelegate dg1 = new MyDelegate(MyMethod);
```

Экземпляр делегата может быть привязан как к статическому, так и к обычному методу класса.

В случае со статическим методом, перед именем метода указывается имя класса. Если имя класса опущено, подразумевается класс, в реализации которого встречается инициализация экземпляра делегата.

В случае с нестатическим методом, перед его именем указывается имя объекта. Если имя объекта опущено, предполагается, что используется текущий экземпляр (`this`):

```
dg1 = new MyDelegate(ClassName.MyMethod);   // статический метод
dg1 = new MyDelegate(ObjectName.MyMethod);  // нестатический метод
```

Экземпляр делегата объявлен и инициализирован. Теперь все готово для его использования.

Вызов метода, на который указывает делегат, напоминает вызов обычного метода, но в данном случае вместо имени метода указывается имя экземпляра делегата:

```
dg1(55);
```

В круглых скобках перечисляется список аргументов. Так как наш метод возвращает строковое значение, мы можем получить его следующим образом:

```
sting str = dg1(55);
```

Давайте рассмотрим простой пример использования делегата (листинг 12.3).

Листинг 12.3. Использование делегата

```
using System;

  //  объявление делегата
delegate string MyDelegate(int arg);

class MyApp
{
   //  статический метод
   public static string MyMethod(int a)
   {
      return a.ToString();
   }
   public static void Main()
   {
      //  создание экземпляра делегата и его инициализация
      MyDelegate dg1 = new MyDelegate(MyApp.MyMethod);
      //  вызов метода, на который указывает делегат
      string r = dg1(2);
      //  вывод полученного значения
      Console.WriteLine(r);
   }
}
```

Рассмотрим типы делегатов.

Типы делегатов

Существует два типа делегатов:

- *одиночные*;
- *комбинированные*.

Пример использования одиночного делегата был нами рассмотрен. Посредством одиночного делегата за один раз может быть вызван единственный метод, в то время как с помощью комбинированного делегата можно вызвать либо один, либо несколько методов одновременно. Различие заключается в сигнатуре делегата, а точнее — в типе возвращаемого значения.

Если делегат объявлен с типом возвращаемого значения `void`, он является комбинированным. Комбинированные делегаты одновременно могут вызвать более одного метода, в то время как сам вызов может возвращать не более одного значения. Комбинированные делегаты наследуют реализацию

от класса System.MulticastDelegate. В случае объявления с типом, отличным от void, делегат является одиночным. В примере из листинга 12.3 был определен делегат с типом возвращаемого значения string. Одиночные делегаты являются производными от System.Delegate.

Рассмотрим видоизмененный пример из листинга 12.3. Текст нового примера приведен в листинге 12.4.

Листинг 12.4. Измененный пример

```
using System;

// объявление делегата
delegate void MyDelegate(int a);     //  тип возвращаемого значения
                                     //  меняется с string на void
class MyApp
{
   //  статический метод
   public static void MyMethod(int a)
   {
      Console.WriteLine(a);
   }
   public static void Main()
   {
      //  создание экземпляра делегата и его инициализация
      MyDelegate dg1 = new MyDelegate(MyApp.MyMethod);
      //  вызов метода, на который указывает делегат
      dg1(3);
   }
}
```

Изменилось объявление делегата с записи:

```
delegate string MyDelegate(int arg);
```

на

```
delegate void MyDelegate(int arg);
```

Откомпилируем и откроем исполняемый файл с помощью IL Disassembler (ildasm.exe). Результат представлен на рис. 12.1.

Как вы можете видеть, делегат с типом возвращаемого значения void расширяет класс System.MulticastDelegate, т. е. является комбинированным делегатом.

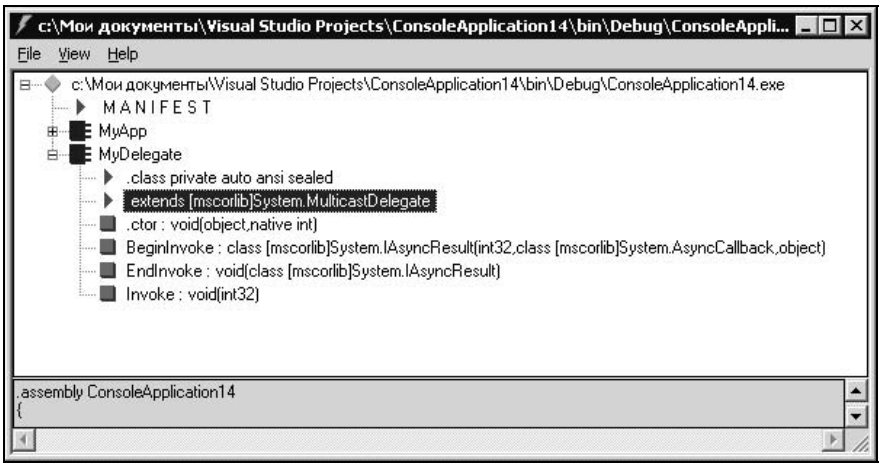


Рис. 12.1. Файл в окне IL Disassembler

Внутренняя структура делегатов

Итак, мы рассмотрели простейшие примеры использования делегатов. Теперь ознакомимся с их внутренней структурой.

Сначала изучим одиночные делегаты, т. к. комбинированные делегаты являются их расширением. Далее мы рассмотрим использование комбинированных делегатов. Отмечу, что все относящееся к одиночным делегатам применимо и к комбинированным. Убедиться в этом можно, посмотрев на иерархию наследования классов `System.Delegate` и `System.MulticastDelegate`.

Приведу объявление обоих классов:

```
public abstract class Delegate : ICloneable, Iserializable
public abstract class MulticastDelegate : Delegate
```

Оба класса являются абстрактными (`abstract`). Их реализация находится в сборке mscorlib.dll пространства имен `System`.

Одиночные делегаты

Одиночные делегаты являются классами, производными от `System.Delegate`. Среди методов и свойств данного класса особенно интересны два свойства: `Target` и `Method`. Приведу их объявления:

```
public object Target {get;}
public MethodInfo Method {get;}
```

Если делегат указывает на нестатический метод, то при его инициализации свойству `Target` присваивается ссылка на экземпляр конкретного объекта — контекст объекта.

В случае со статическим методом его значение `null`. Второе свойство — `Method` — содержит данные о методе. В отличие от первого свойства его значение никогда не равно `null`. Оба свойства являются свойствами только для чтения: вы не можете изменять их значения. Значения данных свойств устанавливаются при инициализации делегата.

Рассмотрим конструктор класса `System.Delegate`. Класс `System.Delegate` имеет два конструктора. Ниже приведены их объявления:

```
protected Delegate
(
   Type target,
   string method
);
```

Данный конструктор предназначен для инициализации делегатов, указывающих на статический метод. Первый аргумент определяет тип (класс), в котором находится статический метод. Второй аргумент задает строку с именем самого метода:

```
protected Delegate
(
   object target,
   string method
);
```

Данный конструктор предназначен для инициализации делегатов, указывающих на нестатический метод. Первый аргумент определяет контекст объекта. Второй аргумент аналогичен второму аргументу в предыдущем конструкторе.

Однако ни один из этих конструкторов вам не понадобится при программировании на C#, поскольку компилятор сам генерирует производный класс делегата и вызывает конструктор, скрывая от вас детали реализации, тем самым, предоставляя более простой синтаксис объявления делегатов.

Рассмотрим, каким образом происходит вызов метода, на который указывает делегат. Для этого вернемся к примеру из листинга 12.3. Встречая следующую строку:

```
string r = dg1(2);
```

компилятор распознает, что `dg1` — это экземпляр делегата. Вызов происходит посредством *рефлексии*.

Рефлексия — это механизм, позволяющий динамически работать с типами, читать информацию о типе и вызывать любые методы класса во время выполнения.

Как вы помните, каждый делегат имеет два свойства: `Method` и `Target`. Свойство `Method` имеет тип `MethodInfo`. В классе `MethodInfo` существует метод `Invoke` (вызвать), позволяющий динамически вызвать метод, данные о котором хранятся в объекте. Компилятор использует значения свойств `Target` и `Method` делегата и генерирует код для вызова `Invoke`. Если вы не знакомы с рефлексией, обязательно с ней познакомьтесь, в будущем вы еще неоднократно с ней встретитесь. Только что вы увидели наглядный пример ее применения. Концепция атрибутов также основана на использовании рефлексии.

Комбинированные делегаты

Рассмотрим пример (листинг 12.5) использования комбинированного делегата.

Листинг 12.5. Использование комбинированного делегата

```
using System;

  //  объявление делегата
delegate void MyDelegate(int a);    //  комбинированный делегат

class MyApp
{
   //  статический метод
   public static void MyMethod(int a)
   {
      Console.WriteLine(a);
   }
   public static void Main()
   {
      //  создание экземпляра делегата и его инициализация
      MyDelegate dg1 = new MyDelegate(MyApp.MyMethod);
      //  вызов метода, на который указывает делегат
      dg1(2);
   }
}
```

Это простейший пример использования комбинированного делегата. Однако в этом случае он идентичен применению одиночного делегата, потому как вызывается единственный метод. А, как вы помните, комбинированные делегаты способны ссылаться на более чем один метод. Для создания комбинированного делегата, который указывает более чем на один метод, необходимо комбинировать делегаты. Для этого в классе `System.Delegate` существует статический метод `Combine`:

```
public static Delegate Combine(Delegate[])
public static Delegate Combine(Delegate, Delegate)
```

Рассмотрим пример (листинг 12.6) использования этого метода.

Листинг 12.6. Использование метода `Combine`

```
using System;

delegate void MyMulticastDelegate(int a);

class MyApp
{
   //  первый метод
   public static void MyM1(int a)
   {
      Console.WriteLine("Метод 1: {0}", a);
   }
   //  второй метод
   public static void MyM2(int a)
   {
      Console.WriteLine("Метод 2: {0}", a);
   }
   public static void Main()
   {
      //  объявление и инициализация делегатов
      //  для первого и второго метода
      MyMulticastDelegate mcdg1 = new
        MyMulticastDelegate(MyApp.MyM1);
      MyMulticastDelegate mcdg2 = new
        MyMulticastDelegate(MyApp.MyM2);
      //  комбинированный делегат будет вызывать оба метода
      MyMulticastDelegate mcdg3;
      //  комбинирование делегатов
      mcdg3 = (MyMulticastDelegate)Delegate.Combine(mcdg1, mcdg2);
      mcdg3(12);
   }
}
```

В данном примере сначала создаются два делегата: `mcdg1` и `mcdg2`. Затем, с помощью метода `Combine` они комбинируются. В итоге получается комбинированный делегат `mcdg3`, результатом вызова которого будет вызов обоих методов: делегата `mcdg1` и `mcdg2`. Чтобы за один вызов `Combine` скомбинировать более двух делегатов, можно воспользоваться перегруженной версией этого метода (листинг 12.7). Вот пример его использования.

Листинг 12.7. Использование перегрузки метода `Combine`

```
using System;

delegate void MyMulticastDelegate(int a);
class MyApp
{
   public static void MyM1(int a)
   {
      Console.WriteLine("Метод 1: {0}", a);
   }
   public static void MyM2(int a)
   {
      Console.WriteLine("Метод 2: {0}", a);
   }
   public static void MyM3(int a)
   {
      Console.WriteLine("Метод 3: {0}", a);
   }
   public static void Main()
   {
      MyMulticastDelegate mcdg1 = new
        MyMulticastDelegate(MyApp.MyM1);
      MyMulticastDelegate mcdg2 = new
        MyMulticastDelegate(MyApp.MyM2);
      MyMulticastDelegate mcdg3 = new
        MyMulticastDelegate(MyApp.MyM3);
      MyMulticastDelegate mcd4;
      //  объявление и инициализация массива делегатов
      Delegate[] dgArr1 = {mcdg1, mcdg2, mcdg3};
      mcd4 = (MyMulticastDelegate)Delegate.Combine(dgArr1);
      mcd4(21);
   }
}
```

Как вы, наверное, заметили, возвращаемое значение метода `Combine` необходимо явно преобразовать к соответствующему типу делегата, т. к. метод возвращает значение с типом `Delegate`:

```
mcdg4 = (MyMulticastDelegate) Delegate.Combine(...);
```

Структура комбинированного делегата

С точки зрения реализации, комбинированный делегат лишь немногим отличается от одиночного.

Различие заключается в следующем: класс `System.MulticastDelegate` содержит дополнительное скрытое поле `_prev`, содержащее ссылку на другой комбинированный делегат.

А сама сущность — комбинированный делегат, представляется в виде связанного списка делегатов с типом возвращаемого значением `void`.

Одиночный и комбинированный делегаты имеют очень много общего, и вы вправе использовать комбинированные делегаты (с типом `void`) как одиночные (см. листинг 12.5).

Дополнительное преимущество комбинированных делегатов в том, что вы можете объединить их в связанный список, используя метод `Combine`, и одновременно вызывать более одного метода. Еще раз посмотрим на метод `Combine`:

```
mcdg3 = (MyMulticastDelegate)Delegate.Combine(mcdg1, mcdg2);
```

Все, что делает данный метод, — это присваивает полю `_prev` делегата `mcdg2` ссылку на делегат `mcdg1` и возвращает ссылку на `mcdg2`. В этом случае делегат `mcdg2` будет являться головным элементом списка. Если же `mcdg1` является связанным списком, то `mcdg2` добавится в этот список.

Рассмотрим еще пару полезных методов. В дополнение к методу `Combine` в классе `System.Delegate` также содержится метод `Remove`:

```
public static Delegate Remove
(
   Delegate source,
   Delegate value
);
```

Данный метод удаляет делегат, заданный в аргументе `value` из связанного списка делегатов, определенного аргументом `source`.

Метод `GetInvocationList`:

```
public virtual Delegate[] GetInvocationList();
```

Данный метод возвращает массив одиночных делегатов. Если метод вызывается для комбинированного делегата, все элементы связанного списка возвращаются в виде одиночных делегатов в массиве. В случае одиночного делегата метод возвращает массив с единственным элементом.

Обратные вызовы

В данном разделе рассмотрим использование делегатов для реализации обратных вызовов. Приведу простой пример (листинг 12.8).

Листинг 12.8. Реализация обратного вызова

```
using System;

delegate int CallBackMethod(int a1, int a2);
class TestApp
{
   public static int MyCallBackMethod(int a1, int a2)
   {
      //  код
      Console.WriteLine("{0}-{1}",a1,a2);
      return 0;
   }
   //  метод в качестве аргумента использует делегат
   public static void SimpleMethod(CallBackMethod cbm)
   {
      // вызов callback-метода
      cbm(20,33);
   }
   public static void Main()
   {
      Console.WriteLine("Обратный вызов метода");
      CallBackMethod callBackMethod = new
      CallBackMethod(MyCallBackMethod);
      SimpleMethod(callBackMethod);
   }
}
```

Для начала необходимо определиться с сигнатурой callback-метода и объявить соответствующий ей делегат (в примере это `CallBackMethod`). Затем реализуется callback-метод `MyCallBackMethod`, на который будет ссылаться делегат. Метод `SimpleMethod` осуществляет обратный вызов посредством делегата `cbm`, который передается в виде аргумента.

Особенностью callback-метода является возможность его передачи в виде аргумента другому методу вместо непосредственного вызова.

События

Событийная модель в CLR строится на основе комбинированных делегатов. В событийной модели используется схема издатель-подписчик.

Класс, предоставляющий события другим классам, способен уведомлять их о переходе в определенное состояние. В свою очередь, желающие получить уведомление обязательно должны зарегистрироваться. Регистрация заключа-

ется в передаче классу-издателю делегата, указывающего на метод — обработчик события. Метод-обработчик находится в классе-подписчике и включает код, который должен быть выполнен в ответ на событие, произошедшее в классе-издателе. Поскольку с делегатами мы разобрались, давайте попробуем предоставить собственную реализацию механизма, обеспечивающего работу событий. Пример обработки событий приведен в листинге 12.9.

Листинг 12.9. Обработка событий

```
using System;

  //  класс делегата
delegate void SimpleMessage(string msg);
  //  класс, предоставляющий события
class ClassWithEvents
{
   //  экземпляр комбинированного делегата
   SimpleMessage subscribers;
   //  метод для регистрации подписчика
   public void AddSubscriber(SimpleMessage sbcr)
   {
      if (subscribers==null) subscribers = sbcr;
      else subscribers = (SimpleMessage)Delegate.Combine
     (subscribers, sbcr);
   }
   //  метод для удаления подписчика
   public void RemoveSubscriber(SimpleMessage sbcr)
   {
      if (subscribers!=null)
      subscribers = (SimpleMessage)Delegate.Remove(subscribers,sbcr);
   }
   //  метод для генерации события
   public void RaiseEvent()
   {
      //  если делегат ссылается хотя бы на один метод, то он вызывается
      if (subscribers!=null) subscribers("Message for subscriber");
   }
}
class MyApp
{
   //  обработчик события 1
   public static void Method1(string msg)
```

```
    {
        Console.WriteLine("Метод1: {0}",msg);
    }
    //  обработчик события 2
    public static void Method2(string msg)
    {
        Console.WriteLine("Метод2: {0}",msg);
    }
    //  обработчик события 3
    public static void Method3(string msg)
    {
        Console.WriteLine("Метод3: {0}",msg);
    }
    public static void Main()
    {
        Console.WriteLine("Делегаты и события");
        //  создание делегатов
        SimpleMessage dg1 = new SimpleMessage(Method1);
        SimpleMessage dg2 = new SimpleMessage(Method2);
        SimpleMessage dg3 = new SimpleMessage(Method3);
        //  создание экземпляра класса-издателя
        ClassWithEvents objectWithEvent = new ClassWithEvents();
        //  регистрация подписчиков
        objectWithEvent.AddSubscriber(dg1);
        objectWithEvent.AddSubscriber(dg2);
        objectWithEvent.AddSubscriber(dg3);
        //  генерация события
        objectWithEvent.RaiseEvent();
        //  удаление подписчиков
        objectWithEvent.RemoveSubscriber(dg3);
        Console.WriteLine("Удаляем подписчика №3");
        objectWithEvent.RaiseEvent();
        objectWithEvent.RemoveSubscriber(dg2);
        Console.WriteLine("Удаляем подписчика №2");
        objectWithEvent.RaiseEvent();
        objectWithEvent.RemoveSubscriber(dg1);
        Console.WriteLine("Удаляем подписчика №1");
        objectWithEvent.RaiseEvent();
    }
}
```

Класс `ClassWithEvents` предоставляет возможность уведомлять подписчиков, посылая им некоторое сообщение. Каждое событие имеет тип, определяю-

щий сигнатуру метода-обработчика события. Этот тип задается делегатом. В классе содержится всего три метода и одно поле. Для подписки на события класс предоставляет метод `AddSubscriber`, а для отписки — `RemoveSubscriber`.

Метод `AddSubscriber` заносит переданный делегат в связанный список. Поле `subscribers` — комбинированный делегат, содержит ссылку на головной элемент связанного списка делегатов подписчиков.

Для генерации события используется метод `RaiseEvent`. В статическом методе `Main` класса `MyApp` приведен код, использующий класс-издатель. Он достаточно прост, потому не буду его комментировать. Данный пример дает хорошее представление о механизме, обеспечивающем работу событий.

Однако в повседневном программировании было бы довольно утомительно для каждого события предоставлять реализацию методов `AddSubscriber` и `RemoveSubscriber`. Поэтому C# поддерживает более простой синтаксис объявления событий. Специально для этого в C# введено ключевое слово `event`. Используя это ключевое слово, событие из рассмотренного примера можно объявить следующим образом:

```
public event SimpleMessage subscribers;
```

Ключевое слово `event` используется для определения экземпляра делегата, посредством которого будут осуществляться вызовы методов — обработчиков событий. Обратите внимание на синтаксис его использования. Вначале идет модификатор доступа события. Ключевое слово `event` свидетельствует о том, что определяется поле — событие класса. Затем указывается тип делегата. Объявление класса делегата должно быть доступно в классе-издателе и в классе-подписчике. В заключение определяется имя события. Заметьте, если класс предоставляет событие другим классам, оно должно быть открытым (`public`). А теперь модифицируем предыдущий пример (листинг 12.9), используя ключевое слово `event`. Текст видоизмененного примера приведен в листинге 12.10.

Листинг 12.10. Модифицированная обработка событий

```
using System;

delegate void SimpleMessage(string msg);
class ClassWithEvents
{
   //  объявление события
   public event SimpleMessage subscribers;
   public void RaiseEvent()
   {
```

```
      //  если с событием сопоставлен хотя бы один метод,
      //  то оно вызывается
     if (subscribers!=null) subscribers("Сообщение для подписчика");
   }
}
class MyApp
{
   public static void Method1(string msg)
   {
      Console.WriteLine("Метод1: {0}",msg);
   }
   public static void Method2(string msg)
   {
      Console.WriteLine("Метод 2: {0}",msg);
   }
   public static void Method3(string msg)
   {
      Console.WriteLine("Метод 3: {0}",msg);
   }
   public static void Main()
   {
      Console.WriteLine("Делегаты и события");
      //  создание делегатов
      SimpleMessage dg1 = new SimpleMessage(Method1);
      SimpleMessage dg2 = new SimpleMessage(Method2);
      SimpleMessage dg3 = new SimpleMessage(Method3);
      ClassWithEvents objectWithEvents = new ClassWithEvents();
      //  регистрирование подписчиков
      //  обратите внимание на синтаксис регистрации
      objectWithEvents.subscribers += dg1;
      objectWithEvents.subscribers += dg2;
      objectWithEvents.subscribers += dg3;
      //  генерация события
      objectWithEvents.RaiseEvent();
      //  удаление подписчиков
      //  обратите внимание на синтаксис удаления
     objectWithEvents.subscribers -= dg3;
     Console.WriteLine("Удаление подписчика №3");
     objectWithEvents.RaiseEvent();
     objectWithEvents.subscribers -= dg2;
     Console.WriteLine("Удаление подписчика №2");
     objectWithEvents.RaiseEvent();
     objectWithEvents.subscribers -= dg1;
     Console.WriteLine("Удаление подписчика №1");
```

```
      objectWithEvents.RaiseEvent();
   }
}
```

Класс `ClassWithEvents` стал проще — в нем нет методов `AddSubscriber` и `RemoveSubscriber`. Более того, также упростился синтаксис подписки и отписки на события. Вместо вызова методов можно использовать перегруженные операторы `+=` и `-=` для подписки и отписки соответственно. Однако ответственность за генерацию события по-прежнему лежит на вас, т. к. метод `RaiseEvent` не изменился. Преимущества данного подхода очевидны.

А теперь посмотрим, что же происходит, когда объявляется событие с ключевым словом `event`. Для этого снова воспользуемся IL Disassembler и откроем последний откомпилированный пример (см. листинг 12.10) (рис. 12.2).

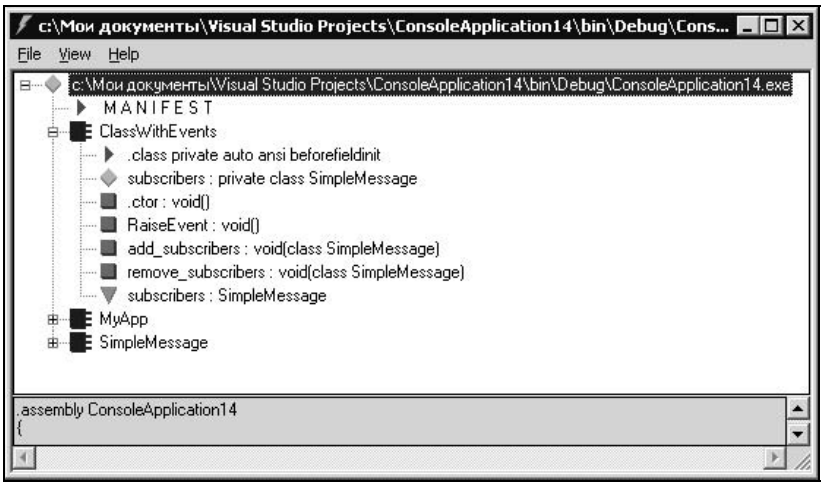


Рис. 12.2. Приложение в программе IL Disassembler

Компилятор объявил скрытое поле-делегат `subscribers`, который содержит список подписчиков. Также компилятор предоставил собственную реализацию методов `add_subscribers` и `remove_subscribers`, причем сигнатуры обоих методов идентичны нашей реализации. А теперь посмотрим на их код.

Примечание

Для удобства чтения установите размер шрифта кода равным 10. Для этого выберите команду меню **View | Set Fonts | Disassembly** и в диалоговом окне **Font** установите в поле **Size** значение `10`.

Дважды щелкните левой кнопкой мыши на имени метода `add_subscribers`, откроется окно, содержащее код реализации метода на IL (Intermediate Language). В этом окне отобразится следующий текст:

```
.method public hidebysig specialname instance void
   add_subscribers(class SimpleMessage 'value') cil managed synchronized
{
   //  Code size    24 (0x18)
   .maxstack 3
   IL_0000: ldarg.0
   IL_0001: ldarg.0
   IL_0002: ldfld   class SimpleMessage ClassWithEvents::subscribers
   IL_0007: ldarg.1
   IL_0008: call    class [mscorlib]System.Delegate
   [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate,
                                      class[mscorlib]System.Delegate)
   IL_000d: castclass SimpleMessage
   IL_0012: stfld   class SimpleMessage ClassWithEvents::subscribers
   IL_0017: ret
}  //  end of method ClassWithEvents::add_subscribers
```

Попробуем разобраться, что же этот метод делает, и восстановить его исходный код. Исходный код можно восстановить так:

```
public void add_subscribers(SimpleMessage sbcr)
{
   subscribers = (SimpleMessage)Delegate.Combine(subscribers, sbcr);
}
```

Как вы видите, он отличается от нашей реализации. Оказывается, что метод `add_subscribers` просто вызывает метод `Combine`. Проверка на `null` оказалась лишней, т. к. метод `Combine` об этом позаботился. Поэтому, если все же придется прибегнуть к использованию собственной реализации данных методов, примените именно такое решение. Теперь попробуем восстановить исходный код `remove_subscribers`:

```
.method public hidebysig specialname instance void
   remove_subscribers(class SimpleMessage 'handler')
   il managed synchronized
{
   //  Code size    24 (0x18)
   .maxstack 8
   IL_0000: ldarg.0
   IL_0001: ldarg.0
   IL_0002: ldfld   class SimpleMessage ClassWithEvents::subscribers
   IL_0007: ldarg.1
   IL_0008: call    class [mscorlib]System.Delegate
   [mscorlib]System.Delegate::Remove(class [mscorlib]System.Delegate,
   class [mscorlib]System.Delegate)
```

```
   IL_000d: castclass SimpleMessage
   IL_0012: stfld   class SimpleMessage ClassWithEvents::subscribers
   IL_0017: ret
}
```

Здесь все аналогично:

```
public void remove_subscribers (SimpleMessage sbcr)
{
   subscribers = (SimpleMessage)Delegate.Remove(subscribers,sbcr);
}
```

Не забывайте использовать именно это решение. А теперь усовершенствуем реализацию примера из листинга 12.10. Текст видоизмененного примера приведен в листинге 12.11.

Листинг 12.11. Усовершенствованный код

```
using System;

delegate void SimpleMessage(string msg);
class ClassWithEvents
{
   SimpleMessage subscribers;
   public void AddSubscriber(SimpleMessage sbcr)
   {
      //  вызов метода Combine
      subscribers = (SimpleMessage)Delegate.Combine(subscribers, sbcr);
   }
   public void RemoveSubscriber(SimpleMessage sbcr)
   {
      //  вызов метода Remove
      subscribers = (SimpleMessage)Delegate.Remove(subscribers,sbcr);
   }
   public void RaiseEvent()
   {
      if (subscribers!=null) subscribers("Сообщение для подписчика");
   }
}
class MyApp
{
   public static void Method1(string msg)
   {
      Console.WriteLine("Метод1: {0}",msg);
   }
```

```
    public static void Method2(string msg)
    {
        Console.WriteLine("Метод 2: {0}",msg);
    }
    public static void Method3(string msg)
    {
        Console.WriteLine("Метод 3: {0}",msg);
    }
    public static void Main()
    {
        Console.WriteLine("Делегаты и события");
        SimpleMessage dg1 = new SimpleMessage(Method1);
        SimpleMessage dg2 = new SimpleMessage(Method2);
        SimpleMessage dg3 = new SimpleMessage(Method3);
        ClassWithEvents objectWithEvent = new ClassWithEvents();
        objectWithEvent.AddSubscriber(dg1);
        objectWithEvent.AddSubscriber(dg2);
        objectWithEvent.AddSubscriber(dg3);
        objectWithEvent.RaiseEvent();
        objectWithEvent.RemoveSubscriber(dg3);
        Console.WriteLine("Удаление подписчика №3");
        objectWithEvent.RaiseEvent();
        objectWithEvent.RemoveSubscriber(dg2);
        Console.WriteLine("Удаление подписчика №2");
        objectWithEvent.RaiseEvent();
        objectWithEvent.RemoveSubscriber(dg1);
        Console.WriteLine("Удаление подписчика №1");
        objectWithEvent.RaiseEvent();
    }
}
```

На этом с внутренней структурой делегатов и событий пока закончу. Далее кратко разберем использование событий на практике (при построении пользовательского интерфейса).

Использование событий

Для начала рассмотрим простейший пример использования событий (его текст приведен в листинге 12.12) при программировании графического интерфейса.

Листинг 12.12. Реализация графического интерфейса с использованием событий

```
using System;
using System.WinForms;
```

```
class MyForm : Form
{
   Button button1 = new Button();
   public MyForm()
   {
      //  установление свойства кнопки
      button1.Text = "Кнопка1";
      //  определение обработчика события
      button1.Click += new EventHandler(button1_Click);
      //  добавление элемента управления в коллекцию формы
      Controls.Add(button1);
   }
   //  обработчик события
   private void button1_Click(object sender, EventArgs e)
   {
      MessageBox.Show("button1_Click");
   }
}
class TestApp
{
   public static void Main()
   {
      Application.Run(new MyForm());
   }
}
```

Класс `Button` содержит событие `Click` (наследуемое от класса `Control`), имеющее тип `EventHandler`. Приведу его объявление:

```
public event EventHandler Click;
```

Его тип такой:

```
public delegate void EventHandler
(
   object sender,
   EventArgs e
);
```

Реализации события содержатся в сборке mscorlib.dll пространства имен `System`.

Как вы видите, делегат является комбинированным, потому как все делегаты, определяющие тип события, должны иметь тип `void`. Первый аргумент содержит ссылку на источник события — объект, генерирующий событие (в нашем случае это форма — `MyForm`). Второй аргумент имеет тип `EventArgs`.

Он содержит аргументы, переданные от издателя подписчику. В нашем случае он не содержит какой-либо полезной информации, поскольку обычно этот класс используется как базовый. Вы вправе определить свой класс, наследующий от `EventArgs`, и внести собственные поля. В классе формы определен метод — обработчик события, соответствующий сигнатуре делегата `EventHandler`, с именем `button1_Click`. Затем, с помощью перегруженного оператора `+=`, регистрируется обработчик события `Click`.

Рассмотрим еще один пример использования событий — событий таймера. Для этого воспользуемся классом, находящемся в пространстве имен `System.WinForms`:

```
public class Timer : Component
```

Класс содержит событие `Tick`,

```
public event EventHandler Tick;
```

которое возникает через определенный период времени, заданный в свойстве `Interval` (время задается в миллисекундах). Для включения и выключения таймера вызываются методы `Start()` и `Stop()` соответственно. После инициализации таймер находится в выключенном состоянии. Пример использования таймера приведен в листинге 12.13.

Листинг 12.13. Использование таймера

```
using System;
using System.WinForms;

class MyForm : Form
{
   Button button1 = new Button();
   Button button2 = new Button();
   Timer timer1 = new Timer();
   public MyForm()
   {
      button1.Text = "Включить таймер";
      button1.Top = 16;
      button1.Left = 16;
      button2.Text = "Выключить таймер";
      button2.Top = 16;
      button2.Left = 96;
      this.Text = "Таймер";
      this.Width = 200;
      this.Height = 88;
      timer1.Interval = 2000;
```

```
      button1.Click += new EventHandler(button1_Click);
      button2.Click += new EventHandler(button2_Click);
      timer1.Tick += new EventHandler(timer1_Tick);
      Controls.Add(button1);
      Controls.Add(button2);
   }
   //  обработчик события кнопки
   private void button1_Click(object sender, EventArgs e)
   {
      timer1.Start();
   }
   //  обработчик события кнопки
   private void button2_Click(object sender, EventArgs e)
   {
      timer1.Stop();
   }
   //  обработчик события таймера
   private void timer1_Tick(object sender, EventArgs e)
   {
      MessageBox.Show("timer1_Tick");
   }
}
class TestApp
{
   public static void Main()
   {
      Application.Run(new MyForm());
   }
}
```

На форме расположены две кнопки: **Включить таймер** и **Выключить таймер**, которые включают и выключают таймер соответственно.

В заключение приведу общий план применения событий. Все, что вам необходимо сделать, — это:

- выбрать событие класса;
- определить его делегат;
- реализовать метод-обработчик, соответствующий делегату;
- зарегистрировать обработчик события.

Особенности использования событий

При использовании событий следует учитывать некоторые особенности. Событие может генерироваться только в методах того класса, в котором оно

определено. Запрещается генерировать событие класса в производных или в других классах. Рассмотрим следующий пример, текст которого приведен в листинге 12.14.

Листинг 12.14. Демонстрация невозможности генерации события

```
using System;

delegate void SimpleMessage(string msg);
  //  базовый класс, содержащий событие
class BaseEventClass
{
   //  событие
   public event SimpleMessage MyEvent;
   //  метод, в котором размещен код генерации события
   public void RaiseEvent(string msg)
   {
      if (MyEvent!=null) MyEvent(msg);    //  код, генерирующий событие
   }
}
  //  производный от BaseEventClass класс наследует событие MyEvent
class DerivedEventClass : BaseEventClass
{
   //  метод, пытающийся сгенерировать событие базового класса
   public void UsingEvent()
   {
      if (MyEvent!=null) MyEvent("Подписчик!");    //  ОШИБКА
   }
}
class Test
{
   public static void Main()
   {
   }
}
```

Несмотря на размер данного примера он достаточно прост. Класс `BaseEventClass` содержит событие. В методе `RaiseEvent` находится код для его генерации. Производный от `BaseEventClass` класс — `DerivedEventClass` в методе `UsingEvent` пытается сгенерировать событие базового класса, однако это ему не удастся, поскольку при компиляции кода выводится следующее сообщение об ошибке:

```
listing14.cs(22,7): error CS0070: The event 'BaseEventClass.MyEvent'
can only appear on the left hand side of += or -= (except when used from
within the class it is defined in)
listing14.cs(22,22): error CS0070: The event 'BaseEventClass.MyEvent'
can only appear on the left hand side of += or -= (except when used from
within the class it is defined in)
```

Перевод:

```
(Событие 'BaseEventClass.MyEvent' может употребляться только с левой
стороны операторов += или -= (кроме тех случаев, когда оно используется
в пределах класса, в котором определено)
```

Как вы видите, этих сообщений два, потому как в коде, генерирующем событие:

```
if (MyEvent!=null) MyEvent("Подписчик!");
```

идентификатор события используется неверным образом два раза. Тот же самый результат вы получите, если попытаетесь поместить код, генерирующий событие в другом классе. Если же вам необходимо генерировать событие из кода других классов (в том числе и производных), предоставьте открытый метод, подобный `RaiseEvent`.

Однако это же сообщение говорит о том, что подписываться на события можно из методов любых классов, в том числе и производных. Следующий пример, текст которого приведен в листинге 12.15, компилируется и работает.

Листинг 12.15. Работающий пример

```
using System;

delegate void SimpleMessage(string msg);

  //  базовый класс, содержащий событие
class BaseEventClass
{
   //  событие
   public event SimpleMessage MyEvent;
   //  метод, в котором размещен код для генерации события
   public void RaiseEvent(string msg)
   {
      if (MyEvent!=null) MyEvent(msg); // код, генерирующий событие
   }
}
  //  производный от BaseEventClass класс наследует событие MyEvent
class DerivedEventClass : BaseEventClass
```

```
{
   //  метод-обработчик события базового класса
   private void DerivedEventClass_MyEvent(string msg)
   {
      Console.WriteLine("DerivedEventClass_MyEvent: {0}",msg);
   }
   //  метод, в котором размещен код подписки на событие базового класса
   public void UsingEvent()
   {
      //  код подписки на событие базового класса
      this.MyEvent += new SimpleMessage(this.DerivedEventClass_MyEvent);
   }
}
class Test
{
   public static void Main()
   {
      DerivedEventClass objWithEvent = new DerivedEventClass();
      objWithEvent.UsingEvent();
      objWithEvent.RaiseEvent("Подписчик!");
   }
}
```

Чтобы подписаться на событие базового класса, используется такая запись:

```
this.MyEvent += new SimpleMessage(this.DerivedEventClass_MyEvent);
```

Как вы видите, задействован текущий экземпляр объекта (`this`), а метод `DerivedEventClass_MyEvent` является не статическим. Однако обычно в таких ситуациях используются виртуальные методы: пример — переопределение виртуального метода `OnPaint`, класса `RichControl` в классе, производном от `Form`.

Почему возникает ошибка? Данный факт объяснить очень просто. Как вы помните, при определении события с именем `MyEvent` компилятор определяет одноименное скрытое поле-делегат. Поскольку генерация события осуществляется посредством делегата, а делегат в свою очередь включает модификатор доступа `private`, то в результате имеется доступ к делегату только из методов класса, в котором определено событие. Так как поддержка делегатов и событий в C# реализована на уровне синтаксиса, в нем существует специальное сообщение об ошибке CS0070, которое предоставляет более ясное пояснение, чем ошибка нарушения доступа к скрытым членам класса, что наверняка сбило бы с толку программиста, не представляющего себе механизм работы событий, реализованный в CLR.

Обработка исключений

В языке C# обработка ошибок во время выполнения приложения основана на понятии *исключений* (exceptions).

Исключение — это особое событие в программе, которое указывает на условие возникновения той или иной ошибки.

При возникновении ошибки инициируется (throw) исключение. Программа может перехватывать (catch) исключения и обрабатывать их.

Блоки *try...catch*

Прежде чем рассматривать обработку исключений в языке C#, рассмотрим пример подпрограммы обработки ошибок, которая написана на языке C:

```
int MyMethod();
...
int result;
result = MyMethod();
if (result != 0)
{
   //  здесь расположены операторы обработки ошибок
}
```

В данном примере используется прототип метода `MyMethod()`, который указывает, что возвращаемое методом значение имеет тип `int`. Прототип содержит код, который обычно является частью подпрограммы.

Переменная `result` принимает значение, которое возвращается методом `MyMethod()`. После чего программа проверяет данное значение на его равенство нулю. Если значение `result` не равно нулю, то произошла ошибка, которая будет обрабатываться в отдельном блоке.

Мы рассмотрели способ обработки ошибок в языке C. Недостатком данного способа является необходимость создания обработчика ошибок для каждого вызова метода. Этот недостаток очень хорошо виден в программах, где имеется несколько вызовов методов. Таким образом код сильно усложняется.

Язык C# позволяет обрабатывать ошибки другим способом, который избавлен от вышеуказанного недостатка.

Основным средством обработки исключений в C# является пара блоков `try ... catch`. Это позволяет отделить обработку ошибок от основного алгоритма программы. Рассмотрим основной синтаксис блоков `try ... catch`:

```
try
{
   //  код, в котором возможно возникновение ошибки
}
```

```
catch (Exception e)
{
   //  код обработки исключительной ситуации
}
```

В листинге 12.16 приведена программа, использующая блоки `try ... catch`.

Листинг 12.16. Простое исключение

```
using System;

public class Exceptions
{
   public static int Main(string[] args)
   {
      byte[] MyStream = new byte[3];
      try
      {
         for (byte b = 0; b < 10; b++)
         {
            Console.WriteLine("Байт {0}: {1}", b+1, b);
            MyStream[b] = b;
         }
      }
      catch (Exception e)
      {
         Console.WriteLine("{0}", e.Message);
      }
      return 0;
   }
}
```

В результате выполнения программы на экран будет выведено следующая информация (рис. 12.3).

Поясним выполнение данной программы. Перед блоком `try` объявлен массив байтов `MyStream` на три элемента. Затем в блоке `try` в цикле последовательно записывается в массив `MyStream` десять байтов. Номер каждого байта и его значение выводится на экран. До четвертого значения переменной цикла программа работает нормально, после чего возникает ошибка выхода за границы массива. Генерируется исключение и управление передается блоку `catch`, который выводит сообщение об ошибке на экран.

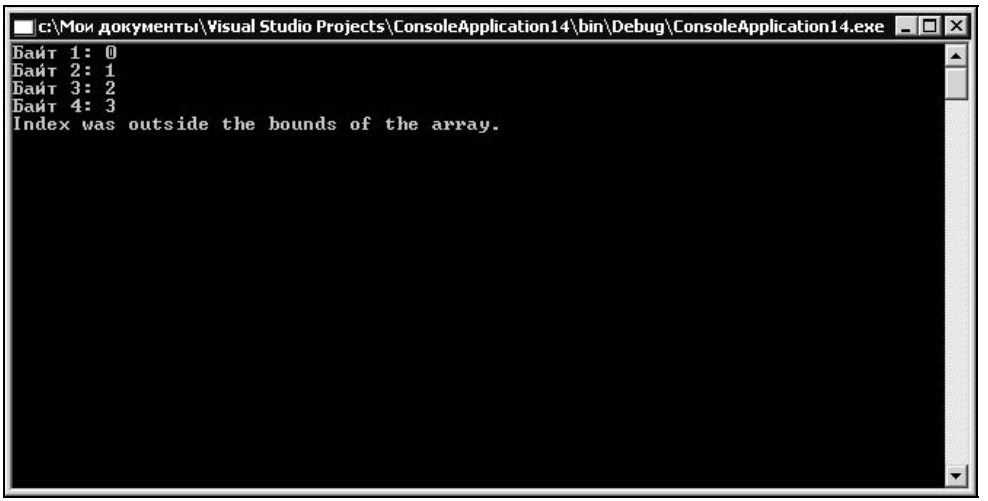


Рис. 12.3. Результат выполнения программы

Блок *finally*

Обычно, после возникновения исключения, программа завершает свое выполнение. Если необходимо высвобождение системных ресурсов, применяется специальный блок `finally`. Данный блок гарантированно будет выполнен до завершения работы метода. Пример использования такого блока приведен в листинге 12.17.

Листинг 12.17. Исключение с блоком `finally`

```
using System;
using System.IO;

public class Exceptions
{
   public static int Main(string[] args)
   {
      byte[] MyStream = new byte[3];
      StreamWriter sw = new StreamWriter("myexcept.txt");

      try
      {
         for (byte b = 0; b < 10; b++)
         {
            sw.WriteLine("Байт {0}: {1}", b+1, b);
```

```
            MyStream[b] = b;
         }
      }
      catch (Exception e)
      {
         Console.WriteLine("{0}", e.Message);
      }
      finally
      {
         sw.WriteLine("Конец");
         sw.Close();
      }
      return 0;
   }
}
```

В данном примере, который является немного видоизмененным примером из листинга 12.16, вывод значений производится не на экран, а в файл. Поэтому при возникновении исключения в блоке `finally` происходит закрытие открытого файла myexcept.txt.

Замечу, что блок `finally` выполняется независимо от того, произошло исключение или нет. Вы можете самостоятельно убедиться в этом факте, если измените строку

```
byte[] MyStream = new byte[3];
```

на

```
byte[] MyStream = new byte[10];
```

и посмотрите содержимое файла myexcept.txt. В нем будет записано слово `Конец` в последней строке.

Способы обработки исключений

Рассмотрим несколько способов обработки исключений, в число которых входят:

1. Обработка нескольких исключений.
2. Обработка и передача исключений.
3. Восстановление нормальной работы после возникновения исключений.

Обработка нескольких исключений

Язык C# позволяет указывать несколько обработчиков исключений для одного блока `try`. Это достигается с помощью добавления дополнительных блоков `catch` после основного блока `try`.

Примечание

Блоки `catch` должны следовать в порядке степени обобщенности исключений, которые они обрабатывают. Иначе компилятор выдаст ошибку.

Приведу пример программы, которая включает в себя два блока `catch` (листинг 12.18).

Листинг 12.18. Использование двух блоков `catch`

```
using System;
using System.IO;

public class Exceptions
{
   public static int Main(string[] args)
   {
      int mySize = 3;
      byte[] MyStream = new byte[mySize];
      int i = 6;
      StreamWriter sw = new StreamWriter("myexcept.txt");

      try
      {
         for (byte b = 0; b < i; b++)
         {
            sw.WriteLine("Байт {0}: {1}", b+1, b);
            MyStream[b] = b;
         }
      }
      catch (IndexOutOfRangeException myEx)
      {
         Console.WriteLine("Индекс недопустим: {0}",myEx.Message);
      }
      catch (Exception e)
      {
         Console.WriteLine("Исключение: {0}", e.Message);
      }
      finally
      {
         sw.WriteLine("Конец");
         sw.Close();
      }
      return 0;
   }
}
```

Так как блок `catch` с исключением `IndexOutOfRangeException` имеет меньшую степень обобщенности, чем блок с исключением `Exception`, то они расположены именно в таком порядке.

Обработка и передача исключений

Одним из методов обработки исключений является метод передачи исключения вызывающей программе. Это можно сделать с помощью оператора `throw`.

Оператор `throw` генерирует указанное исключение и передает его предыдущему блоку `try ... catch`.

В листинге 12.19 приведена программа, использующая оператор `throw`.

Листинг 12.19. Использование оператора `throw`

```
using System;
using System.IO;

public class Exceptions
{
   public static int Main(string[] args)
   {
      Exceptions myE = new Exceptions();
      try
      {
         myE.GenerateException();
      }
      catch (Exception e)
      {
         Console.WriteLine("\n Выполняется главное исключение:");
         while (e != null)
         {
            Console.WriteLine("\t Внутреннее: {0}", e.Message);
            e = e.InnerException;
         }
      }
      finally
      {
         Console.WriteLine("Завершение главного");
      }
      return 0;
   }
```

```
   void GenerateException()
   {
      int mySize = 3;
      byte[] MyStream = new byte[mySize];
      int i = 6;
      StreamWriter sw = new StreamWriter("myexcept.txt");

      try
      {
         for (byte b = 0; b < i; b++)
         {
            sw.WriteLine("Байт {0}: {1}", b+1, b);
            MyStream[b] = b;
         }
      }
      catch (IndexOutOfRangeException myEx)
      {
         Console.WriteLine("Индекс недопустим: {0}",myEx.Message);
         throw new Exception("Из GenerateException", myEx);
      }
       catch (Exception e)
      {
         Console.WriteLine("Исключение: {0}", e.Message);
      }
      finally
      {
         Console.WriteLine("Завершение GenerateException");
         sw.WriteLine("Конец");
         sw.Close();
      }
   }
}
```

Результат выполнения программы представлен на рис. 12.4.

Восстановление нормальной работы после исключений

Часто желательно, чтобы исключения приводили к контролируемому завершению работы программы или к восстановлению нормальной работы программы. В листинге 12.20 приведен пример восстановления нормальной работы программы после возникновения исключения. Программа исправит данные и продолжит дальнейшую работу после исключения.

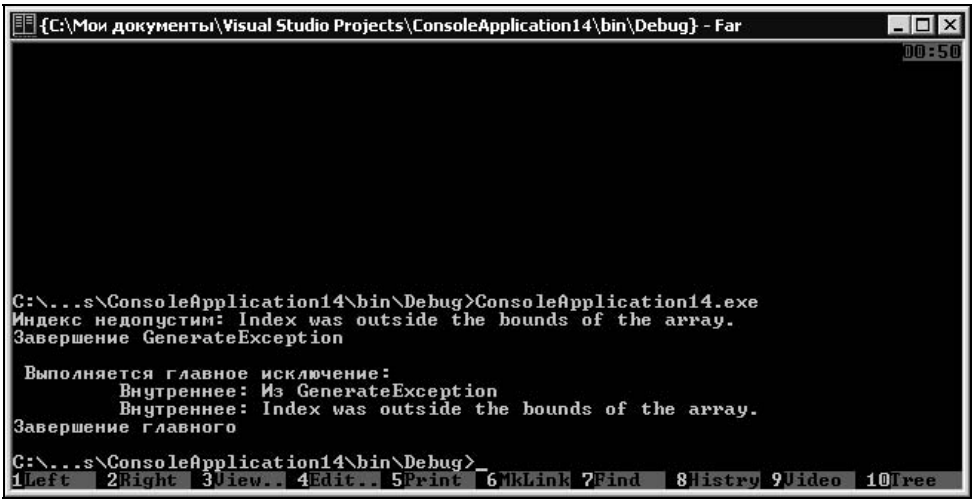


Рис. 12.4. Результат выполнения программы

Листинг 12.20. Восстановление нормальной работы

```
using System;
using System.IO;

public class Exceptions
{
   public static int Main(string[] args)
   {
      Exceptions myE = new Exceptions();
      try
      {
         myE.GenerateException();
      }
      catch (Exception e)
      {
         Console.WriteLine("\n Выполняется главное исключение:");
         while (e != null)
         {
            Console.WriteLine("\t Внутреннее: {0}", e.Message);
            e = e.InnerException;
         }
      }
      finally
      {
         Console.WriteLine("Завершение главного");
      }
```

```
      return 0;
   }
   void GenerateException()
   {
      int mySize = 3;
      byte[] MyStream = new byte[mySize];
      int i = 6;
      do
      {
         StreamWriter sw = new StreamWriter("myexcept.txt");
         try
         {
            for (byte b = 0; b < i; b++)
            {
               sw.WriteLine("Байт {0}: {1}", b+1, b);
               MyStream[b] = b;
            }
            break;
         }
         catch (IndexOutOfRangeException myEx)
         {
            Console.WriteLine("Индекс недопустим: {0}",myEx.Message);
            i--;
         }
         catch (Exception e)
         {
            Console.WriteLine("Исключение: {0}", e.Message);
         }
         finally
         {
            Console.WriteLine("Завершение GenerateException");
            sw.WriteLine("Конец");
            sw.Close();
         }
      }
      while(true);
   }
}
```

Результат работы программы представлен на рис. 12.5.

На этом заканчивается глава и завершается *часть III*. В *части IV* рассматривается работа с данными и изучается технология XML.

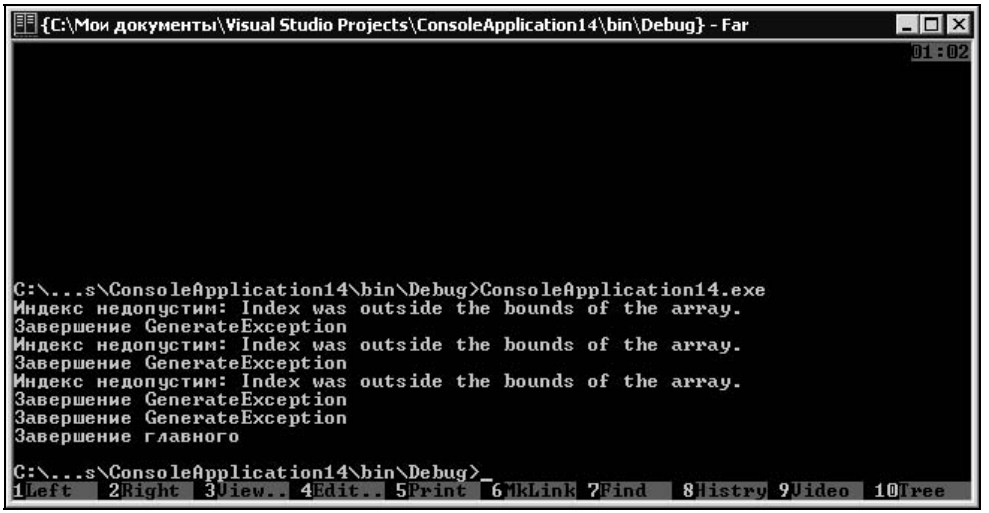
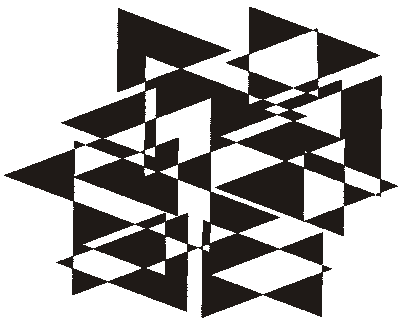


Рис. 12.5. Результат работы программы

ЧАСТЬ IV

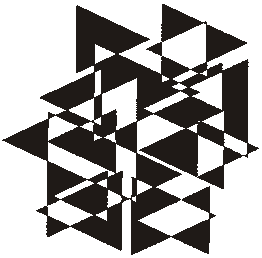


Доступ к данным и использование XML

В этой части рассматриваются две технологии, применяемые в .NET для работы с данными. Это технологии ADO.NET и XML. Технология ADO.NET пришла на смену ранее используемой технологии ADO (ActiveX Data Object) и предоставляет большие преимущества при работе с данными.

Кроме того, затрагиваются вопросы интеграции ADO.NET и XML.

ГЛАВА 13



Технология ADO.NET

Эта глава расскажет вам об архитектуре ADO.NET. Кратко рассматриваются преимущества данной технологии. Вы научитесь работать с наборами данных. Приводится пример создания набора данных и добавления данных в созданные таблицы.

Архитектура ADO.NET

Технология ADO.NET позволяет программам получить доступ к данным, хранящимся на Microsoft SQL Server, или в других хранилищах данных OLE DB. С помощью данной технологии вы можете свободно манипулировать данными.

Технология ADO.NET — это два основных элемента ядра:

- *набор данных* (`DataSet`) — создан для доступа к данным, независимо от их источника. Таким образом, для формирования набора данных могут использоваться несколько различных источников данных, в том числе данные в формате XML. Набор данных состоит из одной или нескольких таблиц данных (`DataTable`), которые, в свою очередь, состоят из *столбцов* (`DataColumn`) и *строк* (`DataRow`) с данными и другой информацией;
- *поставщики данных .NET* (.NET Data Provider) — набор классов, обеспечивающих эффективное манипулирование данными и организацию скоростных каналов передачи данных. Среда .NET Framework поставляется всего с двумя поставщиками данных .NET: SQL Server .NET Data Provider и OLE DB .NET Data Provider. В состав поставщиков данных входят следующие объекты:
  - `Connection` — предназначен для обеспечения соединения с источником данных;

- `Command` — используется для работы с командами базы данных (для чтения данных, изменения данных, запуска хранимых процедур и др.);
- `DataReader` — обеспечивает быстрый однонаправленный (только для чтения) доступ к данным. Данный объект похож на ADO `Recordset`, с серверным, однонаправленным, доступным только для чтения курсором;
- `DataAdapter` — обеспечивает связь между объектом `DataSet` и источником данных. Класс `DataAdapter` использует объект `Command` для выполнения SQL-команд и согласовывает изменения в объекте `DataSet` с источником данных.

Примечание

Вы можете самостоятельно создавать собственные поставщики данных .NET для любых источников данных и затем использовать их в своих приложениях.

На рис. 13.1 схематично представлена архитектура ADO.NET.

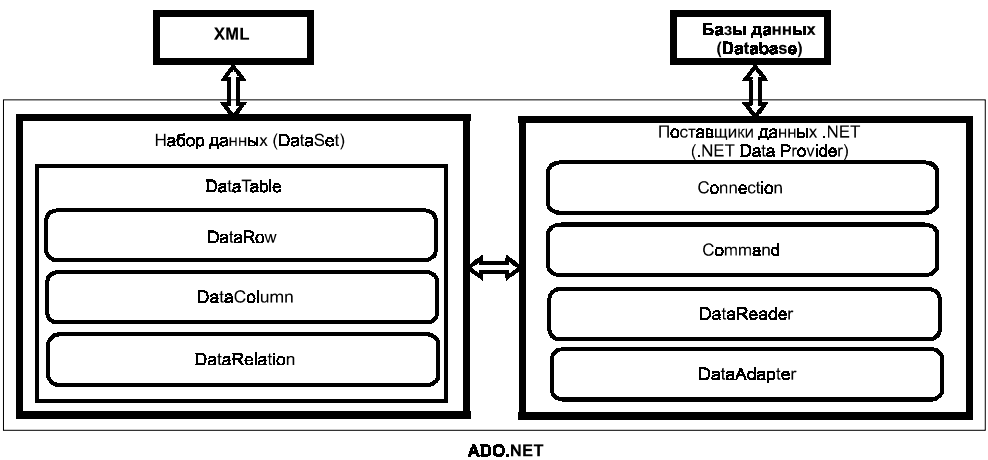


Рис. 13.1. Архитектура ADO.NET

Преимущества ADO.NET

Технология ADO.NET предоставляет ряд преимуществ, в число которых входят:

- *производительность* — улучшенная производительность по сравнению с предыдущими версиями ADO;
- *продуктивность* — предоставление возможности быстрой разработки приложений для доступа к данным с помощью компонентной объектной модели ADO.NET;

- *межплатформенное взаимодействие* — способность взаимодействия с разнородными средами;
- *масштабируемость* — обслуживание большого количества клиентов без ухудшения производительности системы.

Рассмотрим эти преимущества более подробно.

Производительность

В технологии ADO.NET используются отсоединенные наборы данных. За счет этого улучшается производительность и масштабируемость системы.

До создания данной технологии, для обеспечения совместимости типов данных с COM, маршаллинг наборов записей требовал обязательное преобразование типов. Но поскольку теперь отсоединенный набор данных использует формат XML, такое преобразование не нужно.

В технологии ADO.NET сервер баз данных практически не влияет на производительность системы. А поставщики данных .NET обеспечивают неявную организацию пула соединений. Таким образом сокращается время, необходимое для установки соединения с источником данных.

Продуктивность

Как уже отмечалось ранее, технология ADO.NET пришла на смену технологии ADO, и является ее развитием. Поэтому, если вы ранее использовали в своих программах технологию ADO, переход к использованию ADO.NET не будет представлять сложности. Основы остались прежними. Но есть и некоторые отличия, которые значительно увеличивают производительность труда программиста.

В ADO.NET функциональность распределена между всеми объектами значительно лучше, чем в любой из предыдущих версий ADO. В частности, объект `Connection` теперь отвечает только за соединение с источником данных. Если ранее с помощью данного объекта вы могли выполнить запрос, то теперь это сделать нельзя. За осуществление запроса отвечает объект `Command`.

Технология ADO.NET повышает производительность труда программистов также и за счет расширяемости. Все базовые классы ADO.NET представляют собой управляемый код, поэтому вы можете наследовать и расширять эти классы (создавать на их основе свои классы) непосредственно под свои собственные требования.

Для генерации *типизированных наборов данных* (расширенные типы, моделирующие ваши данные) в ADO.NET может использоваться *дизайнер компонентов* (Component Designer). Код, генерируемый дизайнером компонентов,

читается достаточно легко. К тому же, сгенерированные таким образом классы абсолютно безопасны в отношении типов.

Технология ADO.NET улучшает продуктивность за счет расширяемости базовых классов. Кроме того, набор средств, предоставляемых средой Visual Studio .NET, обеспечивает быструю разработку приложений.

Межплатформенное взаимодействие

Для начала рассмотрим проблемы, которые могли возникать ранее, при организации межплатформенного взаимодействия.

Все современные распределенные системы предполагают, что компоненты, необходимые для реализации взаимодействия, используют один и тот же протокол передачи данных и один и тот же формат данных. Таким образом, предполагается такая схема: все компоненты, как передающие, так и получающие данные, должны обязательно использовать технологию COM. Кроме того, передаваемые данные упаковываются в специальный формат *NDR* (Network Data Representation). Из таких NDR-пакетов и формируются потоки между компонентами. Таким образом, возникают две проблемы:

- необходимость присутствия на обеих сторонах (в компонентах принимающих и передающих данные) наличия библиотеки COM;
- сложность в установке и организация взаимодействия через брандмауэры.

То есть, если принимающая и передающая сторона используют технологии фирмы Microsoft — все будет работать нормально. Но если необходимо работать с компонентами других производителей, возникают проблемы.

Технология ADO.NET избавляет вас от этих проблем, т. к. для передачи данных используется универсальный формат XML. Данный формат был выбран потому, что XML — это текстовый формат и он достаточно легок в управлении. Кроме того, XML представляет собой структурированный текст и применение его совместно с протоколом HTTP минимизирует проблемы, связанные с брандмауэрами.

Таким образом, при использовании XML клиентам не нужно присутствия COM. Необходим лишь механизм чтения данных XML, который давно реализован на многих различных платформах, отличающихся от платформ фирмы Microsoft.

Масштабируемость

Обычно, при использовании связи клиент-сервер, клиент устанавливает связь с сервером и не прерывает ее до тех пор, пока не будут выполнены все необходимые запросы. Это вполне разумное решение для небольшого

числа клиентов. Но при достижении определенного числа клиентов сервер становится "узким местом". Открытые соединения с базами данных сильно расходуют ресурсы сети и сервера.

Технология ADO.NET отказывается от применения такой схемы и предлагает использовать отсоединенные наборы данных. В нескольких словах предлагаемую технологию можно описать так: клиент запрашивает необходимые данные у сервера, получает их и разрывает соединение. Таким образом, соединение каждого из клиентов с сервером носит кратковременный характер. Это позволяет производить запрос информации большему количеству клиентов.

Данная методика использовалась и в ранних версиях ADO, но ее реализацией должен был заниматься сам разработчик. В технологии ADO.NET отсоединенные наборы данных применяются всегда.

На первый взгляд может показаться, что использование отсоединенных наборов данных не принесет увеличения производительности, т. к. затраты на установку соединений достаточно высоки. Но в ADO.NET соединение с источником данных автоматически сохраняется в пуле соединений. Поэтому, когда приложение "разрывает" соединение, оно фактически возвращает это соединение в пул. Это позволяет повторно использовать соединение, без дополнительных затрат на его установку.

Работа с ADO.NET

Рассмотрим теперь объекты, входящие в состав ADO.NET, и примеры работы с ними. Все примеры, по возможности, будут на языке C#, который более прост и нагляден, чем C++.

Объект *DataSet*

Если вы уже работали с технологией ADO, то должны знать, что данные передаются в виде *наборов записей* (`RecordSet`).

Набор записей содержит данные в виде таблицы. Независимо от того, из скольких таблиц базы данных взяты данные, они всегда представлены в наборе записей в виде таблицы из столбцов и строк, как будто они все взяты из одной таблицы.

В технологии ADO.NET вместо наборов записей используются *наборы данных* (`DataSet`).

Объект `DataSet` — это размещенное в памяти представление базы данных. Он может содержать внутри себя несколько объектов `DataTable`, представляющих таблицы, и несколько объектов `DataRelation`, описывающих связи и отношения между таблицами.

В более ранних версиях ADO реализовать то, что представляет собой объект `DataSet`, было достаточно тяжело. Можно было использовать несколько объектов `RecordSet` и метод `NextRecordSet()` для переключения между наборами записей. Но описать отношения между наборами записей было невозможно.

Так как технология ADO.NET использует отсоединенные наборы данных, она должна обеспечивать оперативное отслеживание и внесение изменений, произведенных в наборе данных `DataSet`. Объект `DataSet` предоставляет методы, позволяющие согласовывать все изменения, произведенные над данными в наборе данных, с реальным источником данных. Это такие методы, как:

- `AcceptChanges()` — сохраняет все изменения, произведенные в наборе данных, начиная с последнего вызова данного метода, в источнике данных;
- `GetChanges()` — применяется для получения всех измененных данных с момента загрузки набора данных. Сохраняет сведения в другом наборе данных;
- `HasChanges()` — определяет, были ли произведены изменения в наборе данных по сравнению с источником данных;
- `HasErrors()` — определяет, есть ли ошибки в наборе данных;
- `RejectChanges()` — позволяет отклонить все изменения, произведенные над набором данных с момента последнего вызова метода `AcceptChanges()` или с момента получения данных из источника данных.

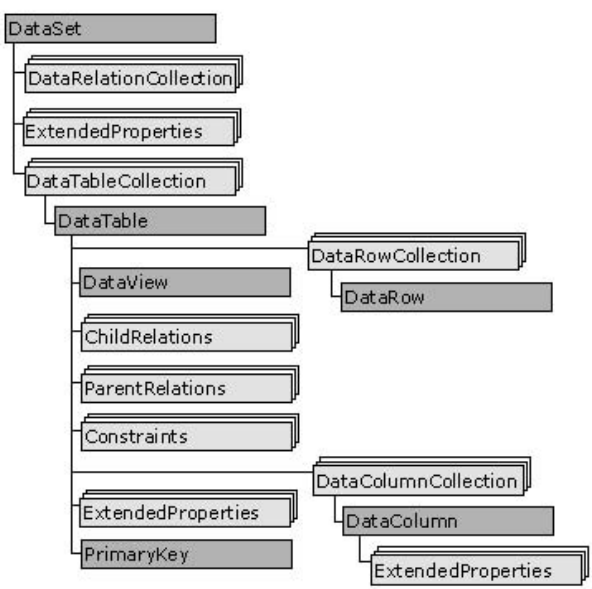


Рис. 13.2. Структура объекта `DataSet`

На рис. 13.2 показана внутренняя структура объекта `DataSet` (данная схема взята из MSDN).

Как видно из рис. 13.2, объект `DataSet` состоит из двух коллекций: `DataRelationCollection` и `DataTableCollection`, а также содержит свойства `ExtendedProperties`:

- коллекция `DataRelationCollection` включает все отношения между таблицами источника данных;
- коллекция `DataTableCollection` содержит набор таблиц, которые входят в набор данных. Каждая из таблиц состоит из коллекций строк (`DataRowCollection`) и столбцов (`DataColumnCollection`), а также других данных, характеризующих каждую таблицу;
- свойства `ExtendedProperties` представляют собой коллекцию свойств, в которые вы можете записывать разнообразную информацию (например, дату и время создания или изменения данных).

Пример создания набора данных

Попробуем создать набор данных `MyDataSet`, который содержит коллекцию из двух таблиц `Customers` и `CDetail`. Затем установим связь между этими таблицами. Схема полей таблиц (в скобках, после названия поля, указан его тип) и связь между ними показаны на рис. 13.3.

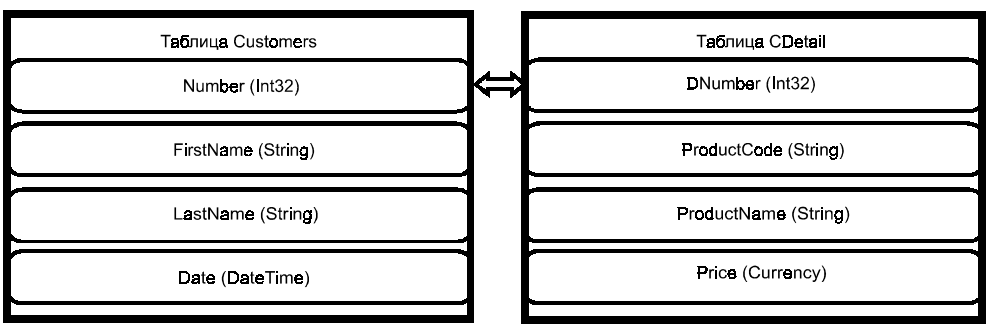


Рис. 13.3. Схема полей создаваемых таблиц

В листинге 13.1 приведен пример динамического создания набора данных, состоящего из двух таблиц: `Customers` и `CDetail`, а также отношений между этими таблицами.

Примечание

Динамическое создание таблиц в данном примере подразумевает создание таблиц непосредственно во время работы приложения. До запуска приложения эти таблицы не существуют.

Листинг 13.1. Динамическое создание набора данных

```
using System;
using System.Data;

  //  создание консольного приложения
namespace ConsoleApplication5
{
   ///  <summary>
   ///  Summary description for Class1.
   ///  </summary>
   class Class1
   {
      ///  <summary>
      ///  The main entry point for the application.
      ///  </summary>
      [STAThread]
      static void Main(string[] args)
      {
         //  создание объекта DataSet
         DataSet MyDataSet = new DataSet("MyDS1");

         //  добавление новой таблицы Customers
         //  в коллекцию таблиц MyDataSet
         MyDataSet.Tables.Add("Customers");

         //  добавление полей (столбцов) таблицы Customers
         MyDataSet.Tables["Customers"].Columns.Add
         ("Number", Type.GetType("System.Int32"));
         MyDataSet.Tables["Customers"].Columns.Add
         ("FirstName", Type.GetType("System.String"));
         MyDataSet.Tables["Customers"].Columns.Add
         ("LastName", Type.GetType("System.String"));
         MyDataSet.Tables["Customers"].Columns.Add
         ("Date", Type.GetType("System.DateTime"));

         //  определение поля Number в качестве первичного ключевого поля
         DataColumn[] keys = new DataColumn[1];
         keys[0] = MyDataSet.Tables["Customers"].Columns["Number"];
         MyDataSet.Tables["Customers"].PrimaryKey = keys;

         //  добавление новой таблицы CDetail
         //  в коллекцию таблиц MyDataSet
         MyDataSet.Tables.Add("CDetail");
```

```
        //  добавление полей (столбцов) таблицы CDetail
        MyDataSet.Tables["CDetail"].Columns.Add
        ("DNumber", Type.GetType("System.Int32"));
        MyDataSet.Tables["CDetail"].Columns.Add
        ("ProductCode", Type.GetType("System.String"));
        MyDataSet.Tables["CDetail"].Columns.Add
        ("ProductName", Type.GetType("System.String"));
        MyDataSet.Tables["CDetail"].Columns.Add
        ("Price", Type.GetType("System.Currency"));

        //  получение объектов типа DataColumn
        //  из двух ранее созданных объектов DataTable
        DataColumn pColumn =
        MyDataSet.Tables["Customers"].Columns["Number"];
        DataColumn chColumn =
        MyDataSet.Tables["CDetail"].Columns["DNumber"];

        //  создание нового отношения между таблицами Customers
        //  и Cdetail, и добавление его в набор данных
        MyDataSet.Relations.Add
       (new DataRelation("Customers_CDetail",pColumn, chColumn));
       MyDataSet.Relations["Customers_CDetail"].Nested = true;
    }
  }
}
```

Итак, экземпляр класса `DataSet` создается с помощью оператора `new`. Затем посредством метода `Add` объекта `Tables` по очереди добавляются одна и вторая таблицы. Добавление столбцов в каждую из таблиц производится также с помощью метода `Add` для коллекции `Columns` объекта `Tables`. Как вы можете заметить, к любой таблице из коллекции таблиц набора данных можно обращаться по ее имени.

Для назначения первичного ключа создается массив `DataColumn` (`keys`), который может содержать одно или несколько ключевых полей. В примере этот массив содержит только одно поле `Number`. Затем массив `keys` присваивается свойству `PrimaryKey` таблицы `Customers`.

Для установления связи между столбцами `Number` и `DNumber` таблиц `Customers` и `CDetail` создается объект `DataRelation` с именем `Customers_CDetail` и добавляется в коллекцию `Relations` набора данных `MyDataSet`.

Наконец, последняя строка в листинге 13.1 указывает на то, что отношение между таблицами будет представлено в виде вложенной структуры. Таким образом будет упрощена работа с этими таблицами в XML-формате.

Теперь попробуем добавить несколько записей в каждую из таблиц. Как это можно сделать, приведено в примере листинга 13.2.

Листинг 13.2. Добавление записей в таблицы набора данных

```
  //  вставка записи в таблицы

  //  первая запись в таблице Customers
DataRow newRow;
newRow = MyDataSet.Tables["Customers"].NewRow();
newRow["Number"] = 1;
newRow["FirstName"] = "Иван";
newRow["LastName"] = "Иванов";
newRow["Date"] = new DateTime(2003, 9, 14);
MyDataSet.Tables["Customers"].Rows.Add(newRow);

  //  вторая запись в таблице Customers
newRow = MyDataSet.Tables["Customers"].NewRow();
newRow["Number"] = 2;
newRow["FirstName"] = "Петр";
newRow["LastName"] = "Петров";
newRow["Date"] = new DateTime(2003, 9, 10);
MyDataSet.Tables["Customers"].Rows.Add(newRow);

  //  первая запись в таблице CDetail
newRow = MyDataSet.Tables["CDetail"].NewRow();
newRow["DNumber"] = 1;
newRow["ProductCode"] = "AB738";
newRow["ProductName"] = "Монитор";
newRow["Price"] = "325.78";
MyDataSet.Tables["CDetail"].Rows.Add(newRow);

  //  вторая запись в таблице CDetail
newRow = MyDataSet.Tables["CDetail"].NewRow();
newRow["DNumber"] = 1;
newRow["ProductCode"] = "BE354";
newRow["ProductName"] = "Принтер";
newRow["Price"] = "123.90";
MyDataSet.Tables["CDetail"].Rows.Add(newRow);

  //  третья запись в таблице CDetail
newRow = MyDataSet.Tables["CDetail"].NewRow();
newRow["DNumber"] = 2;
newRow["ProductCode"] = "F12Y34";
newRow["ProductName"] = "Сканер";
newRow["Price"] = "145.00";
MyDataSet.Tables["CDetail"].Rows.Add(newRow);
```

Текст листинга 13.2 дописывается в конце текста примера, приведенного в листинге 13.1 в процедуре `Main()`.

Таким образом, для первого покупателя Иванова Ивана в таблице `Customers` имеется одна запись, а в таблице `CDetail` — две записи. Для второго покупателя Петрова Петра — в таблице `Customers` и в таблице `CDetail` по одной записи.

Забегая немного вперед, попробуем с помощью XML посмотреть содержимое нашего набора данных.

Примечание

Более подробно XML-документы мы рассмотрим в *гл. 14*.

Объект `DataSet` содержит несколько методов, которые позволяют оперировать с содержимым набора данных, как с файлом в формате XML. Это такие методы, как:

- `ReadXml()` — служит для чтения XML-документов с данными;
- `ReadXmlSchema()` — предназначен для чтения XML-документов со схемами данных;
- `WriteXml()` — выводит как схему, так и данные, находящиеся в таблицах набора данных в виде строк формата XML. В качестве первого аргумента функции указывается поток вывода, а в качестве второго аргумента — объект `XmlWriteMode`, указывающий способ вывода информации;
- `WriteXmlSchema()` — выводит схему набора данных, включая все таблицы и отношения между ними. В качестве первого аргумента функции указывается поток вывода, а в качестве второго аргумента — объект `XmlWriteMode`, указывающий способ вывода информации.

Добавим к программе из листинга 13.2 текст, приведенный в листинге 13.3, который позволит создать XML-документ из набора данных `MyDataSet` и вывести его на экран и в XML-файл.

Листинг 13.3. Создание и вывод на экран и в XML-файл XML-документа

```
  //  создание и вывод на экран набора данных
MyDataSet.WriteXml(Console.Out, XmlWriteMode.WriteSchema);
  //  вывод набора данных в XML-файл
MyDataSet.WriteXml("MyDataSet.xml", XmlWriteMode.WriteSchema);

  //  создание нового набора данных
  //  запись в него данных из XML-файла
DataSet MyDataSet2 = new DataSet("MyDS2");
MyDataSet2.ReadXml("MyDataSet.xml");
```

Две последние строки примера показывают, как можно конвертировать данные из XML-файла в любой набор данных.

Далее приведен текст XML-файла MyDataSet.xml, который также выводится на экран:

```
<?xml version="1.0" standalone="yes"?>
<MyDS1>
 <xs:schema id="MyDS1"
             xmlns=""
             xmlns:xs=http://www.w3.org/2001/X_LSchema
             xmlns:msdata="urn:schemas-m_u99 ?rosoft-com:xml-msdata">
  <xs:element name="MyDS1" msdata:IsDataSet="true"
       msdata:Locale="ru-RU">
   <xs:complexType>
    <xs:choice maxOccurs="unbounded">
     <xs:element name="Customers">
     _ <xs:complexType>
     _  <xs:sequence>
     _    <xs:element name="Number" type="xs:int" />
     _u32 ?  <xs:element name="FirstName" type="xs:string" minOccurs="0" />
     _    <xs:element name="LastName" type="xs:string" minOccurs="0" />
     _    <xs:element name="Date" type="xs:dateTime" minOccurs="0" />
     _u32 ?  <xs:element name="CDetail" minOccurs="0"
                                                 maxOccurs="unbounded">
     _     <xs:complexType>
     _      <xs:sequence>
     _       <xs:element name="DNumber" type="xs:int" minOccurs="0" />
     _u32 ?     <xs:element name="ProductCode" type="xs:string"
                                                       minOccurs="0" />
     _       <xs:element name="ProductName" type="xs:string" minOccurs="0" />
     _u32 ?     <xs:element name="Price"
      msdata:DataType="System.Currency, mscorlib, Version=1.0.5000.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089" type="xs:string"
      minOccurs="0" />
     _      </xs:sequence>
     _     </xs:complexTyp_cf2 >
     _    </xs:element>
     _  </xs:sequence>
     _ </xs:complexType>
     </xs:element>
    </xs:choice>
   </xs:complexType>
   <xs:unique name="Constraint1" msdata:PrimaryKey="true">
    <xs:selector xpath=".//Customers" />
```

```
    <xs:field xpath="Number" />
   </xs:unique>
   <xs:keyref name="Customers_CDetail" refer="Constraint1"
                                                msdata:IsNested="true">
    <xs:selector xpath=".//CDetail" />
    <xs:field xpath="DNumber" />
   </xs:keyref>
  </xs:element>
 </xs:schema>
 <Customers>
  <Number>1</Number>
  <FirstName>Иван</FirstName>
  <LastName>Иванов</LastName>
  <Date>2003-09-14T00:00:00.0000000+08:00</Date>
  <CDetail>
   <DNumber>1</DNumber>
   <ProductCode>AB738</ProductCode>
   <ProductName>Монитор</ProductName>
   <Price>325.78</Price>
  </CDetail>
  <CDetail>
   <DNumber>1</DNumber>
   <ProductCode>BE354</ProductCode>
   <ProductName>Принтер</ProductName>
   <Price>123.90</Price>
  </CDetail>
 </Customers>
 <Customers>
  <Number>2</Number>
  <FirstName>Петр</FirstName>
  <LastName>Петров</LastName>
  <Date>2003-09-10T00:00:00.0000000+08:00</Date>
  <CDetail>
   <DNumber>2</DNumber>
   <ProductCode>F12Y34</ProductCode>
   <ProductName>Сканер</ProductName>
   <Price>145.00</Price>
  </CDetail>
 </Customers>
</MyDS1>
```

Если посмотреть на данный файл, представленный в виде схемы, то мы увидим примерно следующее (рис. 13.4).

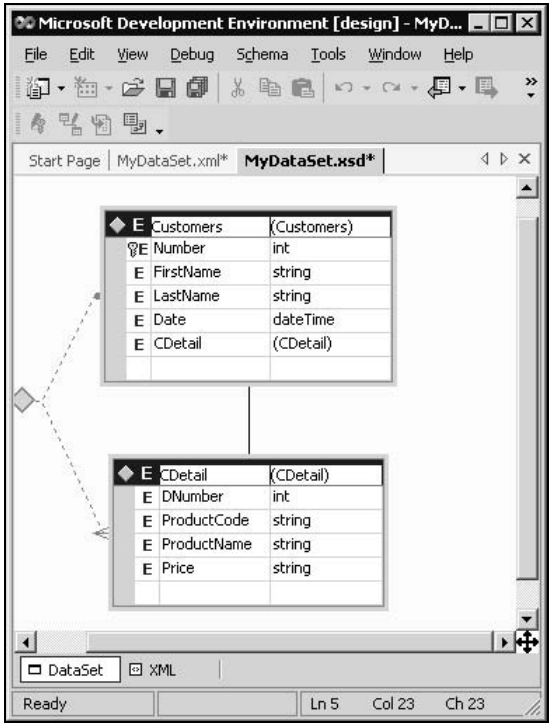


Рис. 13.4. Файл MyDataSet.xml, представленный в виде схемы

Попробуем разобраться с данными в формате XML.

Итак, имя главного (корневого) элемента — `MyDS1` (т. к. таким же было имя набора данных).

Следующий тег `<xs:schema>` содержит определение всех таблиц и отношений между ними в наборе данных `MyDS1`. Все столбцы таблицы `Customers` перечислены в тегах `<xs:element>`. Так как указано, что отношение между таблицами `Customers` и `CDetail` является вложенным, таблица `CDetail` также находится в числе столбцов таблицы `Customers`.

Далее идет тег `<xs:unique>` совместно с атрибутом `msdata:PrimaryKey`. Данный тег используется для определения ключевых полей таблиц, а атрибут обозначает первичный уникальный ключ.

Тег `<xs:keyref>` используется для обозначения внешних ключей (в данном случае — поле `DNumber` таблицы `Customers`, ссылающееся на поле `Number` таблицы `CDetail`).

Затем указаны строки, описывающие данные, находящиеся в таблицах.

Как уже говорилось ранее, набор данных в .NET является отсоединенным от источника данных, поэтому за изменениями в наборе данных вы должны

регулярно следить. Это можно делать с помощью таких методов, как: `GetChanges()`, `HasChanges()` и т. д. Все эти методы уже немного рассматривались в начале данной главы. В примере (листинг 13.4) показано, как можно проверять, были ли сделаны какие-либо изменения в наборе данных, и управлять изменениями.

Листинг 13.4. Работа с изменениями в наборе данных

```
MyDataSet.AcceptChanges();
  //  изменение данных в наборе данных
MyDataSet.Tables["CDetail"].Rows[0]["ProductCode"] = "D7291";
  //  проверка на изменение данных
if(MyDataSet.HasChanges())
   {
      //  получение всех изменений в наборе данных
      //  и помещение их в другой набор данных
      DataSet MyDataSet3 = MyDataSet.GetChanges();
      //  вывод на экран всех изменений
      MyDataSet3.WriteXml(Console.Out, XmlWriteMode.WriteSchema);
      //  подтверждение всех изменений и их внесение в источник данных
      MyDataSet.AcceptChanges();
   }
```

Первый вызов метода `AcceptChanges()` сохраняет все изменения, сделанные в наборе данных на текущий момент. Затем меняется значение поля `ProductCode` в таблице `CDetail`. После чего все сделанные изменения записываются в новый набор данных `MyDataSet3` и выводятся на экран. В результате вы увидите следующую информацию на экране:

```
<?xml version="1.0" standalone="yes"?>
<MyDS1>
 <xs:schema id="MyDS1" xmlns=""
       xmlns:xs=http://www.w3.org/2001/XMLSchema
       xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="MyDS1"
        msdata:IsDataSet="true"
        msdata:Locale="ru-RU">
   <xs:complexType>
    <xs:choice maxOccurs="unbounded">
     <xs:element name="Customers">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Number" type="xs:int" />
        <xs:element name="FirstName" type="xs:string" minOccurs="0" />
```

```
        <xs:element name="LastName" type="xs:string" minOccurs="0" />
        <xs:element name="Date" type="xs:dateTime" minOccurs="0" />
        <xs:element name="CDetail" minOccurs="0" maxOccurs="unbounded">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="DNumber" type="xs:int" minOccurs="0" />
           <xs:element name="ProductCode" type="xs:string" minOccurs="0" />
           <xs:element name="ProductName" type="xs:string" minOccurs="0" />
           <xs:element name="Price" msdata:DataType="System.Currency,
           mscorlib, Version=1.0.5000.0, Culture=neutral,
           PublicKeyToken=b77a5c561934e089" type="xs:string"
           minOccurs="0" />
          </xs:sequence>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:choice>
   </xs:complexType>
   <xs:unique name="Constraint1" msdata:PrimaryKey="true">
    <xs:selector xpath=".//Customers" />
    <xs:field xpath="Number" />
   </xs:unique>
   <xs:keyref name="Customers_CDetail" refer="Constraint1"
                                                msdata:IsNested="true">
    <xs:selector xpath=".//CDetail" />
    <xs:field xpath="DNumber" />
   </xs:keyref>
  </xs:element>
 </xs:schema>
 <Customers>
  <Number>1</Number>
  <FirstName>Иван</FirstName>
  <LastName>Иванов</LastName>
  <Date>2003-09-14T00:00:00.0000000+08:00</Date>
  <CDetail>
   <DNumber>1</DNumber>
   <ProductCode>D7291</ProductCode>
   <ProductName>Монитор</ProductName>
   <Price>325.78</Price>
  </CDetail>
 </Customers>
</MyDS1>
```

Как можно заметить, кроме данных о таблицах и отношениях между ними, в наборе данных `MyDataSet3` имеется запись, в которой произошли изменения.

Последний вызов метода `AcceptChanges()` в листинге 13.4 записывает все изменения в источник данных.

Объект *DataTable*

Мы рассмотрели объект `DataSet` и немного затронули работу с объектом `DataTable`.

Объект `DataTable` представляет собой таблицу, содержащую данные. Как и любая таблица, объект `DataTable` состоит из коллекции строк (`Rows`) и столбцов (`Columns`). Строки представляют собой записи, а столбцы — поля.

Попробуем обратиться к строкам и столбцам таблиц `Customers` и `CDetail`, и выведем названия столбцов, а также содержащиеся в них данные на экран (листинг 13.5).

Листинг 13.5. Вывод таблиц `Customers` и `CDetail` на экран

```
  //  вывод названий столбцов таблицы Customers
  //  и строк, содержащих данные
DataTable MyTable1 = MyDataSet.Tables["Customers"];
  //  названия столбцов
foreach(DataColumn MyColumn in MyTable1.Columns)
{
   Console.Write(MyColumn.ColumnName + "\t");
}
Console.WriteLine("");
  //  строки с данными
foreach(DataRow MyRow in MyTable1.Rows)
{
   //  вывод на экран данных из каждого столбца
   foreach(DataColumn MyColumn in MyTable1.Columns)
   {
      Console.Write(MyRow[MyColumn] + "\t");
   }
   Console.WriteLine("");
}
Console.WriteLine("");

  //  вывод названий столбцов таблицы CDetail и строк, содержащих данные
DataTable MyTable2 = MyDataSet.Tables["CDetail"];
```

```
  //  названия столбцов
foreach(DataColumn MyColumn in MyTable2.Columns)
{
   Console.Write(MyColumn.ColumnName + "\t");
}
Console.WriteLine("");
  //  строки с данными
foreach(DataRow MyRow in MyTable2.Rows)
{
   //  выводим на экран данных из каждого столбца
   foreach(DataColumn MyColumn in MyTable2.Columns)
   {
      Console.Write(MyRow[MyColumn] + "\t");
   }
   Console.WriteLine("");
}
```

В результате выполнения данного фрагмента на экран будет выведена следующая информация:

```
Number        FirstName      LastName          Date
1             Иван           Иванов            14.09.2003 0:00:00
2             Петр           Петров            10.09.2003 0:00:00

DNumber       ProductCode    ProductName       Price
1             D7291          Монитор           325.78
1             BE354          Принтер           123.90
2             F12Y34         Сканер            145.00
```

Добавлять данные в таблицы вы уже умеете. Прежде чем рассказывать о способах удаления данных из таблиц, рассмотрим проблему *отношений* в наборе данных и приведем пример удаления данных.

Отношения в наборе данных

Не зря были созданы две, связанные между собой таблицы набора данных `MyDataSet` в начале главы. Это поможет на примере рассмотреть, каким образом "будут вести себя" таблицы при удалении или изменении данных в главной таблице (в нашем случае — главной является таблица `Customers`).

Свойство `Relations` объекта `DataSet` позволяет устанавливать отношения между таблицами набора данных.

Каждая из таблиц, входящая в отношение, должна "знать" об этом. "Знание" обеспечивают свойства `ChildRelations` и `ParentRelations` объекта `DataTable`. В свойстве `ChildRelations` перечислены все отношения, в которых данная

таблица выступает в роли главной, а в свойстве `ParentRelations` — все отношения, в которых таблица выступает в роли подчиненной.

Кроме установки отношений, вы можете назначить ограничения для этих отношений между таблицами набора данных. Более того, как только создается отношение, среда Visual Studio .NET автоматически создает и ограничения этого отношения.

Существует два типа ограничений:

- `ForeignKeyConstraint` — определяет параметры обновления или удаления данных в связанных таблицах;
- `UniqueConstraint` — при назначении какому-либо полю таблицы такого ограничения значения в данном поле должны быть уникальными.

В табл. 13.1 приведены возможные значения для ограничения `ForeignKeyConstraint`.

Таблица 13.1. *Значения ограничения* `ForeignKeyConstraint`

Значение	Краткое описание
`Cascade`	Все зависимые строки во внешних таблицах будут обновлены или удалены при обновлении или удалении строки в главной таблице (каскадное удаление)
`None`	С зависимыми строками во внешних таблицах не будет производиться никаких действий при обновлении или удалении строки в главной таблице
`SetDefault`	При удалении строки в главной таблице все зависимые строки будут сброшены в значения по умолчанию
`SetNull`	При удалении строки в главной таблице все зависимые строки будут сброшены в значение `DBNull`

Примечание

Все ограничения из табл. 13.1 будут активны только в том случае, если свойство `EnforceConstraint` объекта `DataSet` будет иметь значение `true`.

Рассмотрим пример удаления данных из главной и подчиненной таблицы, если разрешено каскадное удаление (листинг 13.6).

Листинг 13.6. Установка ограничений

```
  //  установка нового ограничения по удалению строк
MyDataSet.Relations["Customers_CDetail"].ChildKeyConstraint.DeleteRule =
       Rule.Cascade;
```

```
  //  вывод в XML-файл данных до удаления
MyDataSet.WriteXml("MyDataSet1.xml");
  //  удаление первой строки в главной таблице
  //  (автоматическое удаление двух строк в подчиненной таблице)
MyDataSet.Tables["Customers"].Rows[0].Delete();
  //  вывод в XML-файл данных после удаления
MyDataSet.WriteXml("MyDataSet2.xml");
```

В результате выполнения программы будет удалена первая запись в главной таблице `Customers` и вместе с ней две записи из подчиненной таблицы `CDetail`. Содержимое файлов MyDataSet1.xml и MyDataSet2.xml приведено в листингах 13.7 и 13.8 (только данные).

Листинг 13.7. Файл MyDataSet1.xml

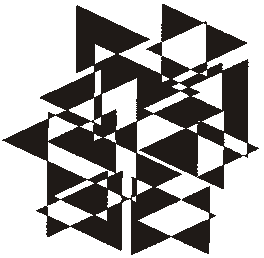
```
<Customers>
 <Number>1</Number>
 <FirstName>Иван</FirstName>
 <LastName>Иванов</LastName>
 <Date>2003-09-14T00:00:00.0000000+08:00</Date>
 <CDetail>
  <DNumber>1</DNumber>
  <ProductCode>AB738</ProductCode>
  <ProductName>Монитор</ProductName>
  <Price>325.78</Price>
 </CDetail>
 <CDetail>
  <DNumber>1</DNumber>
  <ProductCode>BE354</ProductCode>
  <ProductName>Принтер</ProductName>
  <Price>123.90</Price>
 </CDetail>
</Customers>
<Customers>
 <Number>2</Number>
 <FirstName>Петр</FirstName>
 <LastName>Петров</LastName>
 <Date>2003-09-10T00:00:00.0000000+08:00</Date>
 <CDetail>
  <DNumber>2</DNumber>
  <ProductCode>F12Y34</ProductCode>
  <ProductName>Сканер</ProductName>
  <Price>145.00</Price>
 </CDetail>
</Customers>
```

Листинг 13.8. Файл MyDataSet2.xml

```
<Customers>
 <Number>2</Number>
 <FirstName>Петр</FirstName>
 <LastName>Петров</LastName>
 <Date>2003-09-10T00:00:00.0000000+08:00</Date>
 <CDetail>
  <DNumber>2</DNumber>
  <ProductCode>F12Y34</ProductCode>
  <ProductName>Сканер</ProductName>
  <Price>145.00</Price>
 </CDetail>
</Customers>
```

На этом мы заканчиваем данную главу. В *гл. 14* подробно рассматриваются XML-анализаторы и XML-классы.

ГЛАВА 14



Работа с XML-файлами

В этой главе рассматривается популярный формат XML. В начале главы раскрывается понятие XML-анализатора. Далее изучаются основные XML-классы. Кроме того, приводятся примеры работы с файлами XML-формата.

Краткие сведения об XML

Язык *XML* (Extensible Markup Language) похож на язык HTML с одним главным отличием — он может использоваться для описания структурированных данных.

Документы на языке XML представляют собой стандартный переносимый формат для данных, используемых в Web-приложениях, приложениях баз данных и других приложениях, использующих структурные данные.

XML-документы обеспечивают хранение информации в виде текста, которую легко можно найти и отредактировать. Данные в XML-документах хранятся в иерархическом виде, а теги (названные по аналогии с тегами HTML) играют роль описания значения конкретного элемента данных.

В листинге 14.1 приведен пример простого XML-документа.

Листинг 14.1. Простой XML-документ

```
<?xml version="1.0" standalone="yes" ?>
<MyData>
<name>Вячеслав Понамарев</name>
<book>Программирование в Microsoft Visual Studio .NET 2003</book>
<year>2004</year>
<publisher>БХВ-Петербург</publisher>
</MyData>
```

В приведенном примере показаны основные элементы XML-документа. Первая строка определяет версию используемого языка XML (1.0), а также, является ли данный документ одиночным, т. е. не связанным с другими файлами. В данном случае, документ не связан с другим файлом, поэтому параметр `standalone` имеет значение `yes`.

Далее в иерархической форме представлена текстовая информация.

Язык XML представляет собой достаточно мощное средство, поэтому в большинстве продуктов корпорации Microsoft включена поддержка XML. В число продуктов, поддерживающих XML, входят:

- ADO.NET;
- BizTalk;
- SOAP;
- SQL 2000;
- Web Services.

Естественно, этими продуктами поддержка XML не ограничена.

XML-анализаторы

Основная задача XML-анализаторов — программное чтение, просмотр и изменение файлов в формате XML.

Все XML-анализаторы условно делятся на два типа:

- *анализаторы на основе деревьев* — читают XML-файл (или поток) полностью, от начала до конца, и формируют деревья XML-узлов (в качестве XML-узлов выступают XML-теги);
- *анализаторы на основе потоков* — читают XML-поток по мере его получения. Такие анализаторы создаются с одним требованием — они должны удовлетворять спецификации *Simple API for XML* (SAX, "простой API для XML").

Первый тип анализатора преобразует XML-теги в узлы дерева. Таким образом, текст XML-файла, приведенный в листинге 14.1, будет преобразован анализатором в пять узлов: корневой узел `MyData` и узлы `name`, `book`, `year` и `publisher`, расположенные под корневым узлом. Недостаток такого анализатора очевиден — если XML-файл достаточно большой, то и дерево будет очень большим и потребует много оперативной памяти.

Анализатор второго типа в процессе чтения данных генерирует события, которые сообщают приложению о тегах или тексте, которые были прочитаны анализатором. Так как полное дерево не создается, объем памяти, необходимый для такого анализатора, не является большим. Таким образом, данный тип анализатора выгодно использовать для обработки больших по объему XML-файлов.

В среде Microsoft Visual Studio .NET 2003 имеется поддержка как анализаторов на основе деревьев, так и потоковых анализаторов.

Анализаторы на основе деревьев

Рассмотрим несколько объектов, которые входят в состав XML-анализатора на основе дерева, поддерживающего *объектную модель документа* (*DOM*, Document Object Model). Это такие объекты, как:

- `XmlNode`;
- `XmlNodeList`;
- `XmlNamedNodeMap`;
- `XmlDocument`.

Первый объект, который рассмотрим, — это объект `XmlNode`. Он представляет собой один узел XML-документа.

В объектной модели документов почти все объекты являются потомками класса `XmlNode` (к ним относятся `XmlDocument`, `XmlText`, `XmlElement` и др.).

По сути, XML-теги документа соответствуют узлам дерева объектной модели. В листинге 14.2 приведен пример XML-документа (только данные), который уже рассматривался в *гл. 13*.

Листинг 14.2. XML-документ

```
<MyDS1>
 <Customers>
  <Number>1</Number>
  <FirstName>Иван</FirstName>
  <LastName>Иванов</LastName>
  <Date>2003-09-14T00:00:00.0000000+08:00</Date>
  <CDetail>
   <DNumber>1</DNumber>
   <ProductCode>AB738</ProductCode>
   <ProductName>Монитор</ProductName>
   <Price>325.78</Price>
  </CDetail>
  <CDetail>
   <DNumber>1</DNumber>
   <ProductCode>BE354</ProductCode>
   <ProductName>Принтер</ProductName>
   <Price>123.90</Price>
  </CDetail>
 </Customers>
```

```
 <Customers>
  <Number>2</Number>
  <FirstName>Петр</FirstName>
  <LastName>Петров</LastName>
  <Date>2003-09-10T00:00:00.0000000+08:00</Date>
  <CDetail>
   <DNumber>2</DNumber>
   <ProductCode>F12Y34</ProductCode>
   <ProductName>Сканер</ProductName>
   <Price>145.00</Price>
  </CDetail>
 </Customers>
</MyDS1>
```

После преобразования данного XML-документа образуется дерево, представленное на рис. 14.1.

Дерево содержит один корневой узел `MyDS1`, являющийся прямым потомком `XmlNode`. Данный узел имеет тип `XmlDocument`. Под этим корневым узлом расположены два дочерних узла `Customers`, также являющихся потомками `XmlNode`. Тип этих узлов `XmlElement`. Первый узел имеет шесть дочерних узлов, а второй — пять. Узлы относятся к типу `XmlElement`. Первые четыре дочерних узла у каждого из узлов `Customers`, в свою очередь, имеют по одному дочернему элементу типа `XmlText`. Узлы `CDetail` имеют по четыре дочерних узла типа `XmlElement`, каждый из которых, в свою очередь, имеет по одному дочернему элементу типа `XmlText`.

Из рис. 14.1 наглядно видно, как громоздко выглядит дерево даже такого простого и небольшого XML-файла.

Класс `XmlNode` предоставляет методы для создания дерева XML-документа. Это такие методы, как:

- `AppendChild()` — добавляет к данному узлу дочерний узел и помещает его в конец списка дочерних узлов для данного узла;
- `Clone()` — создает точную копию текущего узла;
- `InsertAfter()` — добавляет новый узел сразу после текущего узла;
- `InsertBefore()` — добавляет новый узел перед текущим узлом;
- `PrependChild()` — добавляет к данному узлу дочерний узел и помещает его в начало списка дочерних узлов для данного узла.

Кроме того, класс `XmlNode` имеет несколько свойств, которые помогают находить нужные узлы в дереве и перемещаться по ним. Это такие свойства, как:

- `ChildNodes` — определяет все дочерние узлы данного родительского узла. Может использоваться для перемещения вниз по дереву от корневого узла;

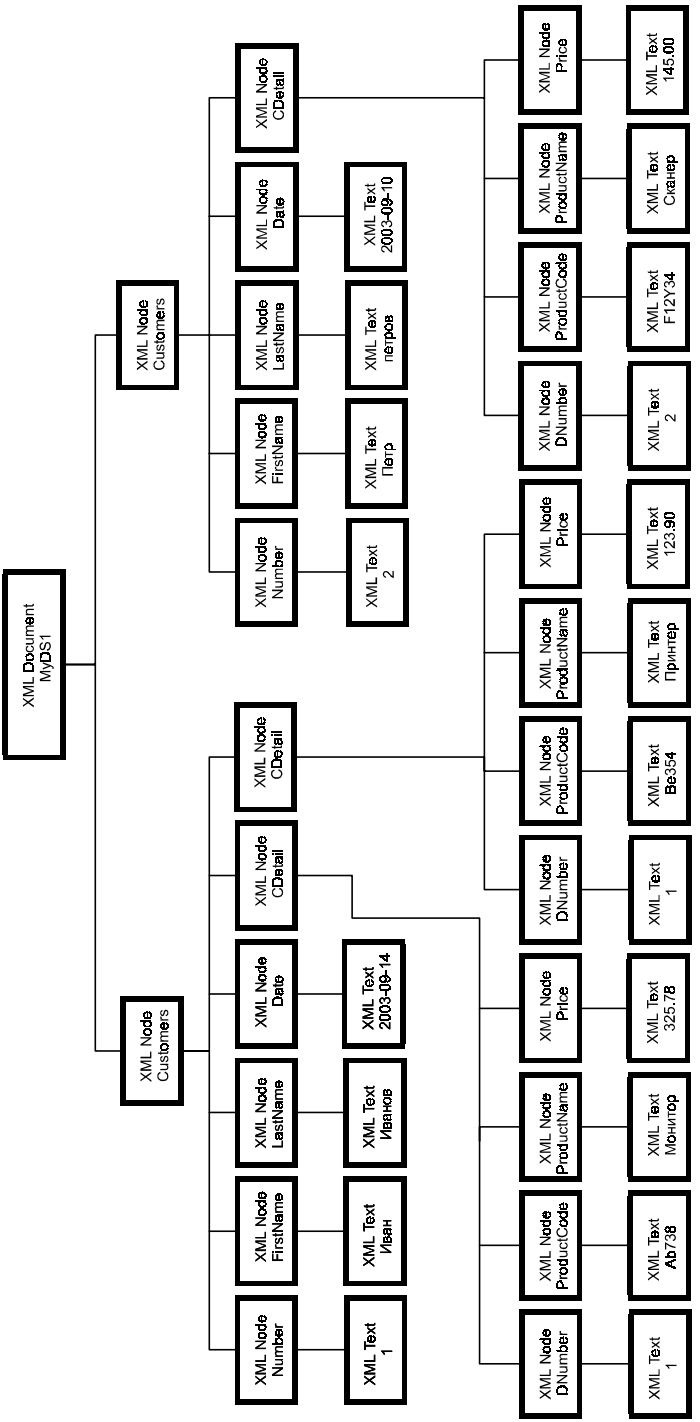


Рис. 14.1. XML-документ, представленный в виде дерева

- `FirstChild` — определяет первый дочерний узел для данного родительского узла;
- `LastChild` — определяет последний дочерний узел для данного родительского узла;
- `NextSibling` — определяет узел, следующий сразу после текущего узла (на одном уровне иерархии);
- `ParentNode` — определяет родительский узел для данного дочернего узла. Может использоваться для перемещения вверх по дереву от дочерних узлов;
- `PreviousSibling` — определяет узел, находящийся непосредственно перед текущим узлом (на одном уровне иерархии).

Следующий объект, который затронем, — это объект `XmlNodeList`.

Объект `XmlNodeList` — это коллекция из одного или нескольких объектов `XmlNode`.

Примечание

В принципе, объект `XmlNodeList` может и не содержать ни одного объекта `XmlNode`.

Тип `XmlNodeList` имеет свойство `ChildNodes` объекта `XmlNode`. Если рассматривать дерево, изображенное на рис. 14.1, то у корневого узла `MyDS1` свойство `ChildNodes` будет коллекцией из двух элементов `XmlNode`. Коллекция построена в виде списка, и вы можете обращаться к любому объекту `XmlNode` по индексу, начиная с нулевого индекса.

Объект `XmlNamedNodeMap` также представляет собой коллекцию объектов `XmlAttribute`, которые являются атрибутами объекта (узла). В рассмотренном примере (см. листинг 14.2) ни у одного узла нет атрибутов. Атрибуты в тексте XML-файла выглядят следующим образом:

```
<Customers category = "Постоянный">
```

В примере атрибут (`XmlAttribute`) узла `Customers` (Заказчики) будет `Постоянный`. И этот узел в дереве иерархии будет располагаться под узлом `Customers`.

Последний объект, относящийся к анализаторам на основе деревьев, который рассмотрим, — это `XmlDocument`. Объект `XmlDocument` представляет собой расширение объекта `XmlNode` и добавляет несколько дополнительных функций.

Объект `XmlDocument` позволяет создавать XML-узлы типа `XmlNode`, `XmlAttribute`, `XmlComment`, `XmlElement` и `XmlText`. Кроме того, объект `XmlDocument` позволяет сохранять и загружать XML-файлы.

В листинге 14.3 приведен пример, как можно программно создать `XmlDocument`. За основу взят уже рассмотренный ранее XML-документ (см. листинг 14.2), для простоты без связанных данных `CDetail`. Пример реализован на языке C#.

Листинг 14.3. Создание нового объекта `XmlDocument`

```
using System;
using System.Xml;

namespace ConsoleApplication6
{
   ///  <summary>
   ///  Summary description for Class1.
   ///  </summary>
   class Class1
   {
      ///  <summary>
      ///  The main entry point for the application.
      ///  </summary>
     [STAThread]
     static void Main(string[] args)
     {
        //  создание нового объекта XmlDocument
        XmlDocument MyXmlDoc = new XmlDocument();
        //  добавление в него нового корневого узла MyDS1
        MyXmlDoc.AppendChild(MyXmlDoc.CreateElement("", "MyDS1", ""));
        XmlElement MyRoot = MyXmlDoc.DocumentElement;
        //  создание новых элементов XmlElement и объекта XmlText
        XmlElement MyCustomers;
        XmlElement MyNumber, MyFirstName, MyLastName, MyDate;
        XmlText MyText;

        //  создание первого узла Customers
        MyCustomers = MyXmlDoc.CreateElement("", "Customers", "");

        //  создание узла Number
        MyNumber = MyXmlDoc.CreateElement("", "Number", "");
        //  создание текста
        MyText = MyXmlDoc.CreateTextNode("1");
        //  добавление текста в качестве дочернего элемента к узлу Number
        MyNumber.AppendChild(MyText);
        //  добавление узла Number в качестве дочернего к узлу Customers
        MyCustomers.AppendChild(MyNumber);
```

```
//  создание узла FirstName
MyFirstName = MyXmlDoc.CreateElement("", "FirstName", "");
//  создание текста
MyText = MyXmlDoc.CreateTextNode("Иван");
//  добавление текста в качестве дочернего элемента
//  к узлу FirstName
MyFirstName.AppendChild(MyText);
//  добавление узла Number в качестве дочернего к узлу Customers
MyCustomers.AppendChild(MyFirstName);

//  создание узла LastName
MyLastName = MyXmlDoc.CreateElement("", "LastName", "");
//  создание текста
MyText = MyXmlDoc.CreateTextNode("Иванов");
//  добавление текста в качестве дочернего элемента
//  к узлу LastName
MyLastName.AppendChild(MyText);
//  добавление узла Number в качестве дочернего к узлу Customers
MyCustomers.AppendChild(MyLastName);

// создание узла Date
MyDate = MyXmlDoc.CreateElement("", "Date", "");
//  создание текста
MyText = MyXmlDoc.CreateTextNode
("2003-09-14T00:00:00.0000000+08:00");
 //  добавление текста в качестве дочернего элемента к узлу Date
 MyDate.AppendChild(MyText);
 //  добавление узла Number в качестве дочернего к узлу Customers
 MyCustomers.AppendChild(MyDate);

 //  добавление созданного узла Customers к корневому узлу
 MyRoot.AppendChild(MyCustomers);

 //  создание второго узла Customers
 MyCustomers = MyXmlDoc.CreateElement("", "Customers", "");

 MyNumber = MyXmlDoc.CreateElement("", "Number", "");
 MyText = MyXmlDoc.CreateTextNode("2");
 MyNumber.AppendChild(MyText);
 MyCustomers.AppendChild(MyNumber);

 MyFirstName = MyXmlDoc.CreateElement("", "FirstName", "");
 MyText = MyXmlDoc.CreateTextNode("Петр");
 MyFirstName.AppendChild(MyText);
 MyCustomers.AppendChild(MyFirstName);
```

```
            MyLastName = MyXmlDoc.CreateElement("", "LastName", "");
            MyText = MyXmlDoc.CreateTextNode("Петров");
            MyLastName.AppendChild(MyText);
            MyCustomers.AppendChild(MyLastName);

            MyDate = MyXmlDoc.CreateElement("", "Date", "");
            MyText = MyXmlDoc.CreateTextNode
            ("2003-09-10T00:00:00.0000000+08:00");
            MyDate.AppendChild(MyText);
            MyCustomers.AppendChild(MyDate);

            MyRoot.AppendChild(MyCustomers);

            // вывод содержимого созданного XML-документа на экран
            Console.WriteLine(MyXmlDoc.InnerXml);
            //  вывод содержимого созданного XML-документа в файл
            MyXmlDoc.Save("MyDSNew.xml");
        }
    }
}
```

В результате, созданный XML-документ (листинг 14.3) будет отображен на экране и выведен в файл MyDSNew.xml. Содержимое файла приведено в листинге 14.4.

Листинг 14.4. XML-файл, созданный в результате работы программы

```
<MyDS1>
 <Customers>
  <Number>1</Number>
  <FirstName>Иван</FirstName>
  <LastName>Иванов</LastName>
  <Date>2003-09-14T00:00:00.0000000+08:00</Date>
 </Customers>
 <Customers>
  <Number>2</Number>
  <FirstName>Петр</FirstName>
  <LastName>Петров</LastName>
  <Date>2003-09-10T00:00:00.0000000+08:00</Date>
 </Customers>
</MyDS1>
```

Вы можете не только сохранять содержимое объекта `XmlDocument` в XML-файл, но и загружать содержимое XML-файла в объект `XmlDocument`. Для проверки, добавьте в конце процедуры `Main()`, приведенной в листинге 14.3, следующие строки:

```
  //  создание нового объекта XmlDocument
XmlDocument MyXmlDoc2 = new XmlDocument();
  //  загрузка содержимого файла MyDSNew.xml в данный объект
MyXmlDoc2.Load("MyDSNew.xml");
  //  демонстрация того, что данные были загружены
Console.WriteLine(MyXmlDoc2.InnerXml);
```

Анализаторы на основе потоков

Анализаторы такого типа представлены в Microsoft Visual Studio .NET 2003 двумя основными объектами:

- `XmlReader`;
- `XmlWriter`.

Рассмотрим их по порядку.

Объект `XmlReader` предоставляет быстрый "не кэшируемый" однонаправленный доступ к потоковым XML-данным. Есть два производных класса от объекта `XmlReader`:

- класс `XmlTextReader` — предназначен для чтения данных из XML-текста;
- класс `XmlNodeReader` — служит для чтения потока узлов из объекта `XmlDocument`.

Оба объекта, `XmlReader` и `XmlWriter`, читают XML-файл по одному тегу из потока за один раз. Поток может начинаться как в начале XML-файла (при чтении всего XML-документа), так и с определенного узла (XML-документ читается частично).

Рассмотрим на небольшом примере, как можно прочитать нужные узлы и их атрибуты из XML-файла, а остальные данные просто игнорировать. Для этого используем XML-файл, приведенный в листинге 14.5.

Листинг 14.5. XML-файл MyOrders.xml

```
<MyOrders>
 <Order number="6372">
  <Product number="213" name="Монитор" price="364.90">
   Монитор Samsung
  </Product>
  <Product number="113" name="Принтер" price="216.00">
   Принтер Canon
  </Product>
  <Product number="145" name="Сканер" price="283.20">
   Сканер Umax
  </Product>
```

```
 </Order>
 <Order number="6328">
  <Product number="213" name="Монитор" price="364.90">
   Монитор Samsung
  </Product>
  <Product number="012" name="Клавиатура" price="16.00">
   Клавиатура Microsoft
  </Product>
  <Product number="015" name="Мышь" price="13.20">
   Мышь Microsoft
  </Product>
 </Order>
</MyOrders>
```

Рассмотрим программу (на языке C#), которая обрабатывает XML-файл и выводит на экран следующие данные: номер заказа (атрибут `number` тега `Order`) и номер товара (атрибут `code` тега `Product`).

Пусть XML-документ, приведенный в листинге 14.5, записан в файл с именем MyOrders.xml. В листинге 14.6 приведен текст программы обработки XML-файла.

Листинг 14.6. Потоковая обработка XML-файла

```
using System;
using System.IO;
using System.Xml;

namespace ConsoleApplication7
{
   ///  <summary>
   ///  Summary description for Class1.
   ///  </summary>
   class Class1
   {
      ///  <summary>
      ///  The main entry point for the application.
      ///  </summary>
      [STAThread]
      static void Main(string[] args)
      {
         Class1 MyTest = new Class1();
         StreamReader MyStream = new StreamReader("MyDS1.xml");
         XmlTextReader MyTR = new XmlTextReader(MyStream);
```

```
            while(MyTR.Read())
            {
               if(MyTR.NodeType == XmlNodeType.Element && MyTR.Name
               == "Order")
               {
                  MyTest.MyProc1(MyTR);
               }
            }
            Console.ReadLine();
         }

         public void MyProc1(XmlTextReader reader)
         {
            Console.WriteLine("Заказ № " + reader.GetAttribute("number"));
            while(!(reader.NodeType == XmlNodeType.EndElement
            && reader.Name == "Order") && reader.Read())
            {
               обработка данных заказа
               if(reader.NodeType == XmlNodeType.Element
               && reader.Name == "Product")
               {
                  Console.WriteLine("Номер товара:" +
                  reader.GetAttribute("number"));
               }
            }
         }
      }
}
```

Поясним текст программы. Вначале создается объект `MyTR` типа `XmlTextReader`. Далее вызываем в цикле метод `Read()`, который последовательно читает теги XML-документа до самого конца. Как только встречается тег, имеющий тип `XmlElement` с именем `Order`, вызывается процедура `MyProc1`, которая будет обрабатывать заказ. Процедура `MyProc1` читает и обрабатывает в цикле все теги внутри тега `Order` и выводит необходимую информацию, если такая встречается. Как только появляется завершающий тег `Order`, происходит выход из процедуры `MyProc1`.

В результате выполнения программы (листинг 14.6) на экран будет выведена информация, представленная на рис. 14.2.

Мы рассмотрели пример использования объекта `XmlTextReader`. Применение объекта `XmlNodeReader` аналогично, с одним отличием. Данный объект читает узлы из полного дерева XML. Таким образом, при чтении больших XML-файлов с помощью `XmlNodeReader` могут возникнуть некоторые трудности.

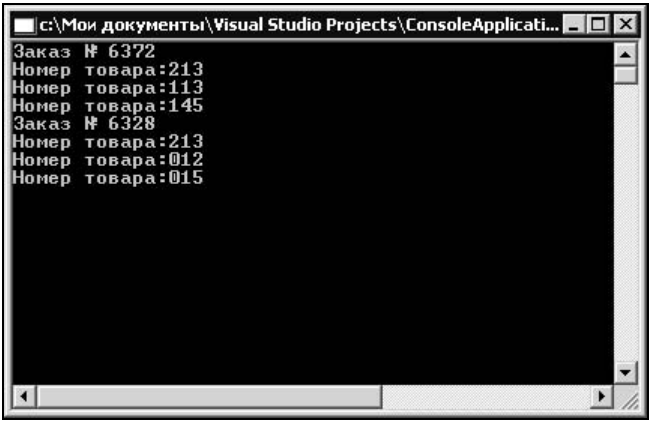


Рис. 14.2. Результат выполнения программы

Объект `XmlWriter` предоставляет быстрый "не кэшируемый" способ записи потоковых XML-данных. В отличие от объекта `XmlReader`, объект `XmlWriter` имеет всего один производный объект `XmlTextWriter`.

Рассмотрим на примере, как можно использовать объект `XmlTextWriter` для создания небольшого XML-файла (листинг 14.7).

Листинг 14.7. Создание XML-файла с помощью объекта `XmlTextWriter`

```
using System;
using System.Xml;

namespace ConsoleApplication8
{
   ///  <summary>
   ///  Summary description for Class1.
   ///  </summary>
   class Class1
   {
      ///  <summary>
      ///  The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
      {
         //  создание нового объекта MyWriter типа XmlTextWriter,
         //  кодировка Unicode
         XmlTextWriter MyWriter = new XmlTextWriter("MyTest.xml",
         new System.Text.UnicodeEncoding());
```

```
            //  задание форматирования строк в документе
            MyWriter.Formatting = Formatting.Indented;
            //  задание четырех пробелов для форматирования
            MyWriter.Indentation = 4;
            //  вывод информации о документе (версия XML и др.)
            MyWriter.WriteStartDocument();
            //  вывод комментария о документе
            MyWriter.WriteComment("Это пример создания XML-файла");
            //  вывод стартового тега
            MyWriter.WriteStartElement("Order");
            //  вывод атрибутов тега
            MyWriter.WriteStartAttribute("number", "number", "number");
            MyWriter.WriteString("6327");
            //  завершение вывода атрибутов
            MyWriter.WriteEndAttribute();
            //  новый тег Product
            MyWriter.WriteStartElement("Product");
            //  текст внутри тега Product
            MyWriter.WriteString("Монитор Samsung");
            //  конец тега Product
            MyWriter.WriteEndElement();
            //  конец тега Order
            MyWriter.WriteEndElement();
            //  конец XML-документа
            MyWriter.WriteEndDocument();
            //  сброс всего содержимого из буфера
            MyWriter.Flush();
            //  закрытие потока
            MyWriter.Close();
        }
    }
}
```

В результате выполнения программы сформируется файл MyTest.xml, текст которого приведен в листинге 14.8.

Листинг 14.8. Содержимое файла MyTest.xml

```
<?xml version="1.0" encoding="utf-16"?>
<!--Это пример создания XML-файла-->
<Order number:number="6327" xmlns:number="number">
  <Product>Монитор Samsung</Product>
</Order>
```

Основные свойства и методы объекта `XmlTextWriter`, некоторые из которых использовались в программе листинга 14.7, перечислены в табл. 14.1.

Таблица 14.1. *Основные свойства и методы объекта* `XmlTextWriter`

Свойство/Метод	Краткое описание
`Formatting`	Определяет, будет ли отформатирован XML-файл при создании. По умолчанию имеет значение `None` (не форматированный файл)
`Indentation`	Определяет, сколько символов будут использоваться для отступов при форматировании XML-файла
`IndentChar`	Определяет, какой символ будет использоваться для отступов при форматировании XML-файла. По умолчанию этот символ — пробел
`Close()`	Служит для закрытия текущего потока записи
`Flush()`	Сбрасывает все данные из временного буфера в файл потока
`WriteAttributes()`	Служит для записи атрибутов текущего тега
`WriteComment()`	Позволяет записать комментарий в XML-файл
`WriteEndAttribute()`	Завершает запись атрибутов тега
`WriteEndDocument()`	Завершает все открытые теги и их атрибуты. Возвращает объект `XmlTextWriter` в исходное (стартовое) состояние
`WriteEndElement()`	Завершает текущий открытый тег
`WriteStartAttribute()`	Начинает запись атрибутов тега
`WriteStartDocument()`	Записывает в XML-файл описание документа, включая номер версии XML (на текущий момент это версия 1.0), использующегося в данном файле
`WrtiteStartElement()`	Начинает запись нового тега
`WriteString()`	Позволяет записать текстовое содержимое открытого тега или атрибута

Преобразование XML-формата

Для преобразования формата XML в формат HTML, который используется в Web-браузерах, среда разработки предоставляет программисту объект `XslTransform`.

XSL (Extensible Stylesheet Language) — это расширяемый язык таблиц стилей. Он применяется для преобразования XML в HTML посредством объекта `XslTransform`. С помощью специально создаваемых шаблонов XSL вы можете просматривать XML-файлы в окнах Web-браузеров, таких как Internet Explorer, Netscape Navigator и др. В листинге 14.9 приведен XSL-шаблон для преобразования XML-файла MyDSNew.xml (см. листинг 14.4).

Листинг 14.9. XSL-шаблон для преобразования файла MyDSNew.xml

```
<xsl:stylesheet version="1.0"
   xmlns:xls="http://www.w3.org/1999/XSL/Transform">
<xsl:template match = "/">
<html>
<head><title>Список клиентов</title></head>
<style>
.hdr{background-color=#ffeedd; font-weight=bold;}
</style>
<body>
<B>Список клиентов</B>
<table style="border-collapse:collapse" border="1">
<tr>
   <td class="hdr">Номер</td>
   <td class="hdr">Имя</td>
   <td class="hdr">Фамилия</td>
   <td class="hdr">Дата</td>
</tr>
<xsl:for-each select="//MyDS1/Customers">
<tr>
   <td><xsl:value-of select="Number"/></td>
   <td><xsl:value-of select="FirstName"/></td>
   <td><xsl:value-of select="LastName"/></td>
   <td><xsl:value-of select="Date"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

В листинге 14.10 приведен текст консольного приложения, которое преобразует файл MyDSNew.xml в файл Customers.html, используя XSL-шаблон.

Листинг 14.10. Преобразование XML-файла в HTML-файл

```
using System;
using System.Xml;
using System.Xml.Xsl;
using System.Xml.XPath;
using System.IO;

namespace ConsoleApplication9
{
   ///  <summary>
   ///  Summary description for Class1.
   ///  </summary>
   class Class1
   {
      ///  <summary>
      ///  The main entry point for the application.
      ///  </summary>
      [STAThread]
      static void Main(string[] args)
      {
         //  создание нового объекта XmlDocument
         XmlDocument MyXmlDoc = new XmlDocument();
         //  загрузка содержимого файла MyDSNew.xml в данный объект
         MyXmlDoc.Load("MyDSNew.xml");
         //  создание нового объекта XslTransform
         XslTransform MyXSL = new XslTransform();
         //  загрузка содержимого XSL-шаблона
         MyXSL.Load("MyTemplate.xsl");
         //  создание нового объекта XmlTextWriter
         XmlTextWriter MyWriter = new XmlTextWriter
         ("Customers.html",null);
         //  преобразование XML в HTML
         MyXSL.Transform(MyXmlDoc, null, MyWriter);
         //  закрытие потока записи
         MyWriter.Close();
         //  вывод содержимого файла Customers.html на экран
         StreamReader MyStream = new StreamReader("Customers.html");
         Console.Write(MyStream.ReadToEnd());
      }
   }
}
```

На рис. 14.3 показан файл Customers.html, отображенный в окне браузера Internet Explorer.

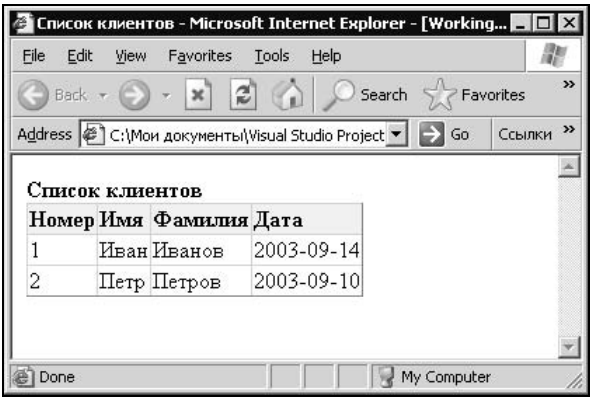
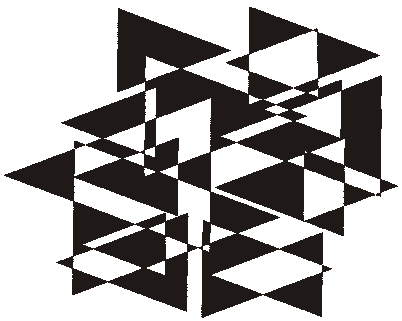


Рис. 14.3. Вид файла Customers.html в окне Internet Explorer

В данной главе мы рассмотрели объекты, предоставляемые средой Microsoft Visual Studio .NET 2003, для работы с файлами в формате XML. Эти сведения, а также сведения из *гл. 13* понадобятся вам для успешного усвоения материала следующей *части V*. В ней рассказывается о создании Web-приложений с помощью среды Visual Studio .NET 2003. Вы познакомитесь с Web-службами и Web-сервисами.

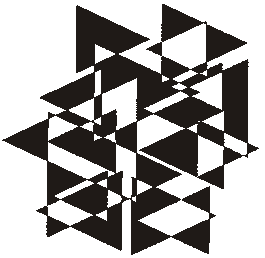
ЧАСТЬ V



Создание Web-приложений в Visual Studio .NET 2003

В этой части рассматриваются принципы построения Web-приложений на языках C++ и C# в среде Visual Studio .NET 2003. Раскрываются понятия Web-служб. Описывается технология ASP.NET. На одном большом примере рассматривается принцип создания Web-магазина, занимающегося продажей компьютерной техники через Интернет. Изучается отладка готовых приложений с помощью встроенного в Visual Studio .NET 2003 компилятора.

ГЛАВА 15



Принципы создания Web-приложений

В данной главе вы познакомитесь с Web-службами, которые обеспечивают доступ к программным компонентам, используя стандартные для Интернета протоколы HTTP и SMTP. Рассмотрим внутреннюю архитектуру Web-служб. На небольших примерах показано, как можно создать Web-службу, и как созданная Web-служба будет использоваться Web-клиентами. В конце главы описывается прием внедрения в Web-службу поддержки безопасности на уровне системы и на уровне приложения.

Понятие Web-службы

Понятие *служба* (service) в широком смысле обозначает получение какой-либо услуги от *поставщика* услуг. Например, вы получаете от химчистки услугу по чистке предметов. Таким образом, становитесь *потребителем* услуги, а химчистка — поставщиком услуги.

Обычно, программное обеспечение, которое используется в работе, приобретается на каком-либо носителе (CD, DVD и т. д.). Но представим себе такой вариант. У вас есть какое-либо электронное устройство, которое может подключаться к Интернету и работать с ним. С помощью данного устройства вы подключаетесь к Интернету и указываете устройству, чтобы оно использовало службу Outlook для получения списка ваших контактов. Для этого не требуется устанавливать на устройство саму программу Outlook. После работы со списком контактов вы сохраняете изменения на сервере. Кроме того, безопасность ваших данных будет гарантирована сервером данных.

Такие возможности предоставляют *Web-службы* (Web Services).

Архитектура Web-служб

Web-службы представляют собой распределенные программные компоненты, которые доступны через стандартные протоколы, использующиеся в Интернете.

Как можно заметить, распределенными программными компонентами также являются компоненты COM, но отличие Web-служб от компонентов COM заключается в их доступности через интернет-протоколы.

Компоненты COM могут работать только с другими компонентами COM. А Web-службы позволяют программному обеспечению работать с любым приложением, которое понимает язык XML и способно принять XML-поток с помощью протокола HTTP. Как вы теперь понимаете, язык XML является основой, которая используется в Web-службах.

.NET Web Services состоит из нескольких компонентов.

- *Форматы связи Web-служб.* Это технология, которая позволяет определить, каким образом производится обмен данными между поставщиком и потребителем услуги, а также определить формат данных для запроса и отклика.
- *Описание Web-службы на языке WSDL.* Язык Web Services Description Language (*WSDL*, язык описания Web-службы) применяется для описания способа применения службы. Его можно представить себе как инструкцию по использованию данной службы.
- *Открытие Web-службы.* Реклама службы для облегчения ее поиска в Интернете.

Рассмотрим эти компоненты более подробно.

Форматы связи Web-служб

Web-службы .NET Web Services в настоящее время основаны на трех основных протоколах: *HTTP GET*, *HTTP POST* и *SOAP*. Это стандартные протоколы передачи данных для сети Интернет.

Протоколы HTTP GET и HTTP POST используют в качестве основы протокол HTTP (HyperText Transfer Protocol, протокол передачи гипертекста). Методы `GET` и `POST` данного протокола используются в различных серверных архитектурах. Они кодируют параметры HTTP-запроса в виде пар *имя-значение*.

Метод `GET` генерирует строку HTTP-запроса и добавляет ее к URL сценария, который обрабатывает запрос на сервере.

Метод `POST` передает пары имя-значение в теле HTTP-запроса.

Протокол SOAP (Simple Object Access Protocol, простой протокол доступа к объектам) по своим действиям похож на протоколы HTTP GET и HTTP

POST, но клиенты, использующие протокол SOAP, отправляют сообщения-запросы серверу в формате XML, а не HTTP. Сервер также получает отклик в формате XML.

Так как данные отправляются в формате XML, они имеют лучшую структуру и могут передавать более сложную информацию. Кроме того, SOAP может работать с другими протоколами, такими как *SMTP* (Simple Mail Transfer Protocol, простой протокол передачи почты).

Описание Web-службы на языке WSDL

Для того чтобы клиенты Web-служб могли взаимодействовать с ней, должно существовать описание методов или интерфейсов, которые поддерживаются данной Web-службой. Такое описание создается в XML-схеме, которая носит название WSDL.

Как известно, для описания компонентов COM применяются библиотеки типов и IDL-сценарии. WSDL-описания, так же как и IDL-сценарии, описывают вызовы методов интерфейса в виде списка входных и выходных параметров для каждого из методов. Основное отличие IDL-сценария от WSDL-файла заключается в том, что все описания в файле WSDL представлены на языке XML.

В листинге 15.1 приведен пример простого WSDL-файла (пример взят из документации MSDN).

Листинг 15.1. Простой WSDL-файл

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="FooSample"
targetNamespace="http://tempuri.org/wsdl/"
xmlns:wsdlns="http://tempuri.org/wsdl/"
xmlns:typens="http://tempuri.org/xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:stk="http://schemas.microsoft.com/soap-toolkit/wsdl-extension"
xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
<schema targetNamespace="http://tempuri.org/xsd"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
elementFormDefault="qualified" >
</schema>
</types>
```

```
<message name="Simple.foo">
<part name="arg" type="xsd:int"/>
</message>

<message name="Simple.fooResponse">
<part name="result" type="xsd:int"/>
</message>

<portType name="SimplePortType">
<operation name="foo" parameterOrder="arg" >
<input message="wsdlns:Simple.foo"/>
<output message="wsdlns:Simple.fooResponse"/>
</operation>
</portType>

<binding name="SimpleBinding" type="wsdlns:SimplePortType">
<stk:binding preferredEncoding="UTF-8" />
<soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="foo">
<soap:operation
soapAction="http://tempuri.org/action/Simple.foo"/>
<input>
<soap:body use="encoded" namespace="http://tempuri.org/message/"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</input>
<output>
<soap:body use="encoded" namespace="http://tempuri.org/message/"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</output>
</operation>
</binding>

<service name="FOOSAMPLEService">
<port name="SimplePort" binding="wsdlns:SimpleBinding">
<soap:address location="http://carlos:8080/FooSample/FooSample.asp"/>
</port>
</service>
</definitions>
```

Как можно заметить, данный файл полностью реализован на языке XML. Рассмотрим общую структуру WSDL-файла с тем, чтобы понять суть приведенного WSDL-файла.

Главным (или корневым) элементом WSDL-файла является элемент `<definitions>`. Он открывает WSDL-файл. Тег `</definitions>` закрывает

WSDL-файл. Внутри данного элемента содержится непосредственное описание Web-службы. Для этого используются следующие элементы:

- `<types>` — содержит описания типов;
- `<message>` — сообщение, служит для описания параметров, которыми обмениваются поставщики и клиенты Web-службы. Каждый метод, описываемый в WSDL-файле, имеет два сообщения (элемента `<message>`): *входное* и *выходное*. Входное сообщение описывает параметры метода, а выходное — возвращаемые методом данные. Каждое такое сообщение может содержать параметры `<part>`. Любой из параметров связан с конкретным типом, указанным в элементе `<types>`;
- `<portType>` — представляет собой абстрактный набор операций, которые поддерживаются Web-службой;
- `<operation>` — операция. Абстрактное описание действия, которое поддерживается Web-службой. Каждая операция содержит входные и выходные значения, которые задаются с помощью элементов `<input message>` и `<output message>`;
- `<binding>` — спецификация протокола и формата данных определенного типа порта. Содержит операции, входные и выходные данные для каждой операции;
- `<service>` — коллекция портов;
- `<port>` — включает описание используемого протокола и спецификации формата данных, а также сетевой адрес, к которому могут обращаться клиенты Web-службы.

В листинге 15.2 представлена общая структура типичного WSDL-файла.

Листинг 15.2. Общая структура WSDL-файла

```
<definitions name="..." targetNamespace="..." xmlns:...>

<types>
...
</types>

<message name="...">
...
</message>
...
<portType name="...">
<operation name="...">
<input message="..."/>
```

```
<output message="..."/>
</operation>
...
</portType>
...
<binding name="...">
<protocol:binding ...>
<operation name="...">
<protocol:operation ...>
<input>
...
</input>
<output>
...
</output>
</operation>
...
</binding>
...
<service name="...">
<port name="..." binding="...">
<protocol:address location="..."/>
</port>
...
</service>
</definitions>
```

Как уже говорилось ранее, в элементе `<types>` содержится описание физического типа, который определен в XML-схеме.

Приведу пример создания небольшого WSDL-файла по частям.

Для каждого метода Web-службы данного порта определяются два сообщения (`<message>`): входное и выходное. Если Web-служба поддерживает все три протокола, то элементов `<message>` будет шесть. По два на каждый из протоколов. Существует такое соглашение о присвоении имен:

```
Имя_метода + Протокол + In (Out)
```

Примечание

Такое соглашение действует и при автоматической генерации WSDL средой Visual Studio .NET 2003.

То есть, если для метода `GetName()` существует поддержка всех трех протоколов, этот метод будет иметь шесть сообщений, текст которых приведен в листинге 15.3.

Листинг 15.3. Сообщения для метода Web-службы

```
<message name="GetNameSoapIn">...</message>
<message name="GetNameSoapOut">...</message>
<message name="GetNameHttpGetIn">...</message>
<message name="GetNameHttpGetOut">...</message>
<message name="GetNameHttpPostIn">...</message>
<message name="GetNameHttpPostOut">...</message>
```

Для каждого протокола, который поддерживается Web-службой, необходимо описать по одному элементу `<portType>`. Все операции в этих элементах определяются с помощью элементов `<operation>`. Существует такое соглашение для элементов `<portType>`:

```
Имя_Web-службы + Протокол
```

Дополню пример, приведенный в листинге 15.3, описанием типов портов `<portType>`, связанных с Web-службой `MyWS` (листинг 15.4).

Листинг 15.4. Описание типов портов Web-службы

```
<portType name="MyWSSoap">
<operation name="GetName">
<input message="GetNameSoapIn"/>
<output message="GetNameSoapOut"/>
</operation>
</portType>

<portType name="MyWSHttpGet">
<operation name="GetName">
<input message="GetNameHttpGetIn"/>
<output message="GetNameHttpGetOut"/>
</operation>
</portType>

<portType name="MyWSHttpPost">
<operation name="GetName">
<input message="GetNameHttpPostIn"/>
<output message="GetNameHttpPostOut"/>
</operation>
</portType>
```

Описанные типы портов представляют собой абстрактные операции для каждого порта. Кроме типов портов, для каждого протокола необходимо добавить по одному элементу `<binding>`, который предоставляет конкретную информацию об используемых протоколах, способе передачи данных и ме-

стонахождении Web-службы. В листинге 15.5 приведены описания элементов <binding>.

Листинг 15.5. Описание привязок Web-службы

```
<binding name="MyWSSoap" type="s0:MyWsSoap">
<soap:binding style="document"
     transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="GetName">
<soap:operation soapAction="http://tempuri.org/GetName"
     style="document"/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>

<binding name="MyWSHttpGet" type="s0:MyWsHttpGet">
<http:binding verb="GET"/>
<operation name="GetName">
<http:operation location="/GetName"/>
<input>
<http:urlEncoded/>
</input>
<output>
<mime:mimeXml part="Body"/>
</output>
</operation>
</binding>

<binding name="MyWSHttpPost" type="s0:MyWsHttpPost">
<http:binding verb="POST"/>
<operation name="GetName">
<http:operation location="/GetName"/>
<input>
<mime:content type="application/x-www-form-urlencoded"/>
</input>
<output>
<mime:mimeXml part="Body"/>
</output>
</operation>
</binding>
```

Итак, для протокола SOAP из листинга 15.5 была задана привязка `<soap:binding>`. В роли транспорта выступают SOAP-сообщения, которые передаются с помощью протокола HTTP. Элемент `<soap:operation>` задает HTTP-заголовок `soapAction`, который указывает на метод. Входными и выходными данными (`<input>` и `<output>`) для SOAP являются SOAP-сообщения.

Для протоколов HTTP GET и HTTP POST заданы привязки `<http:binding>` и команды `GET` и `POST`. Кроме того, задан URL, который указывает на метод. В примере из листинга 15.5 это элемент `<http:operation>`, который содержит атрибут `location`, равный `/GetName`. Отличие протоколов HTTP GET и HTTP POST заключается в способе передачи параметров Web-серверу. Протокол HTTP GET посылает параметры в строке запроса, а HTTP POST — в виде данных формы. Поэтому элементы `<input>` для этих протоколов различаются.

Осталось определить порты, которые поддерживаются Web-службой `MyWS`. Для этого используется элемент `<service>`. А для каждого поддерживаемого протокола добавляется один элемент `<port>` внутри элемента `<service>`. В листинге 15.6 приведено продолжение примера из листинга 15.5.

Листинг 15.6. Определение портов, поддерживаемых Web-службой

```
<service name="MyWS">

<port name="MyWSSoap" binding="s0:MyWSSoap">
<soap:address location="http://localhost/MyWS/MyWS.asmx"/>
</port>

<port name="MyWSHttpGet" binding="s0:MyWSHttpGet">
<soap:address location="http://localhost/MyWS/MyWS.asmx"/>
</port>

<port name="MyWSHttpPost" binding="s0:MyWSHttpPost">
<soap:address location="http://localhost/MyWS/MyWS.asmx"/>
</port>

</service>
```

Открытие Web-службы

Все Web-службы делятся на *закрытые* и *открытые*.

Некоторые службы не рекламируются публично. Такие Web-службы относятся к *закрытым*. Они применяются в бизнесе для предоставления права пользования только своим бизнес-партнерам.

Для рекламирования своих Web-служб (которые относятся к *открытым*), их авторы размещают в сети Интернет файлы открытий (файлы с расширением disco). Эти файлы указывают, где находятся реальные Web-службы и их описания. Кроме того, disco-файлы могут содержать ссылки на другие disco-файлы. Будущие клиенты Web-служб могут просматривать такие файлы. Процесс поиска Web-службы и ее описания называется *Web Service discovery* или *открытие Web-службы*.

Существует два способа открытия Web-служб: *статическое открытие* и *динамическое открытие*. Рассмотрим их подробно.

При статическом открытии (static discovery) явно создается файл открытия. В этом файле содержится указание на WSDL.

Примечание

При создании Web-службы средствами Microsoft Visual Studio .NET disco-файл создается средой автоматически.

Типичный пример статического disco-файла приведен в листинге 15.7.

Листинг 15.7. Статический disco-файл

```
<?xml version="1.0" ?>
<disco: discovery xmlns=disco"http://schemas.xmlsoap.org/disco/"
                  xmlns:scl="http://schemas.xmlsoap.org/disco/scl/">
<scl:contractRef ref="http://localhost1/MyWS1.asmx?WSDL"/>
<scl:discoveryRef ref="http://localhost2/MyWS2.disco"/>
</disco:discovery>
```

Внутри корневого элемента `<discovery>`, который описан в пространстве имен `http://schemas.xmlsoap.org/disco`, может быть один или несколько элементов `<contractRef>` или `<discoveryRef>`, принадлежащие пространству имен `http://schemas.xmlsoap.org/disco/scl`.

Тег `<contractRef>` используется для указания ссылки на реальный URL Web-службы, который должен содержать WSDL или описание Web-службы. Теги `<discoveryRef>` содержат ссылки на другие disco-документы. В примере (листинг 15.7) имеются оба тега. Первый ссылается на Web-службу, а второй — на другой disco-документ. Таким способом можно создавать структурированные сети связанных disco-документов.

Вы можете отказаться от явного указания адресов URL для Web-служб. Вместо этого можно разрешить динамическое открытие (dynamic discovery), которое автоматически формирует список всех Web-служб, поддерживаемых данным сайтом. Таким образом, для сайта можно распределить связанные Web-службы по разным папкам, а затем предоставить в каждой папке по одному динамическому disco-файлу.

Пример простого динамического disco-файла приведен в листинге 15.8 (взят из документации MSDN).

Листинг 15.8. Простой динамический disco-файл

```
<?xml version="1.0" encoding="utf-8" ?>
<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>
```

Корневым тегом любого динамического disco-файла является тег `<dynamicDiscovery>`.

Тегами `<exclude path>` вы можете указать исключаемые пути. Таким образом, динамический механизм не будет осуществлять поиск Web-служб в данных папках, входящих в папку, где расположен динамический disco-файл. Общий синтаксис данного тега следующий:

```
<exclude path="Исключаемая_папка" />
```

Поставщики Web-служб

Поставщики Web-служб предоставляют и рекламируют свои услуги так, чтобы клиенты смогли этими услугами воспользоваться. Так как Web-службы работают на основе протокола HTTP, на компьютере поставщика Web-служб должен обязательно работать какой-либо Web-сервер, поддерживающий протокол HTTP (Internet Information Server (IIS), Apache и т. д.).

Дальнейшие примеры будут приводиться использованием IIS, т. к. он пока является единственным, поддерживаемым средой .NET.

Попробуем создать Web-службу с именем `MyWS`, которая позволит клиентам получать информацию из базы данных `MyDataBase` (эта база данных должна быть создана). Доступ к данным будем производить на основе технологии ADO.NET, которая рассматривалась в *части IV*. Для этого выберем в диалоговом окне создания нового проекта (**New Project**) пиктограмму **ASP.NET Web Service** (рис. 15.1).

Напомню, что для создания Web-службы необходимо, чтобы на вашем компьютере был установлен Internet Information Server. В противном случае создать Web-службу невозможно.

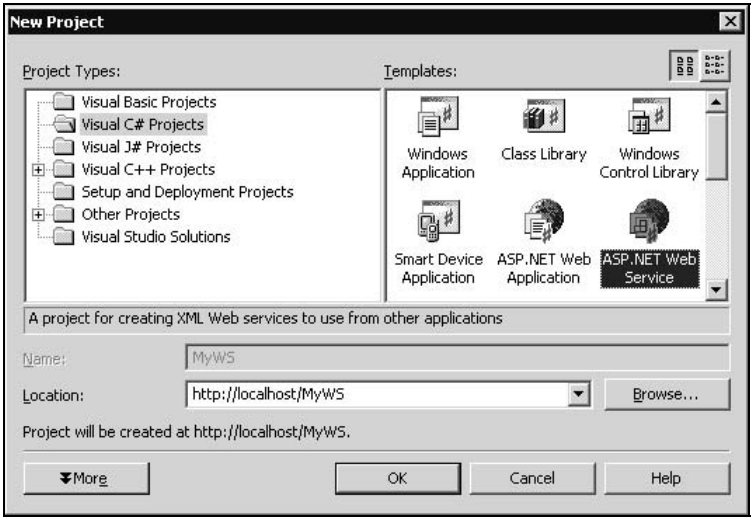


Рис. 15.1. Создание новой Web-службы

В листинге 15.9 приведен текст примера простой Web-службы, написанного на языке C#. Данная Web-служба с именем `MyWS` предоставляет Web-клиентам два метода, позволяющие получить список всех клиентов, а также осуществить поиск нужного клиента по его фамилии.

Листинг 15.9. Простая Web-служба

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Web;
using System.Web.Services;

namespace MyWS
{
   [WebService(Namespace="http://localhost/")]
   public class Service1 : System.Web.Services.WebService
   {
      private static string MyConnectStr = "provider=sqloledb;
      server=(local); database=MyDataBase; Integrated Security=SSPI";
      [WebMethod(Description="Возвращает список всех клиентов")]
      public DataSet GetClients()
      {
         OleDbDataAdapter MyDBAdapter;
         DataSet MyDataSet;
```

```
            MyDBAdapter = new OleDbDataAdapter
            ("select * from LastName", MyConnectStr);
            MyDataSet = new DataSet();
            MyDBAdapter.Fill(MyDataSet,"LastName");
            return MyDataSet;
        }
        [WebMethod]
        public DataSet GetLastName(string MyStr)
        {
            OleDbDataAdapter MyDBAdapter;
            DataSet MyDataSet;
            MyDBAdapter= new OleDbDataAdapter("select * from
            LastName where LastName='" + MyStr + "'", MyConnectStr);
            MyDataSet = new DataSet();
            MyDBAdapter.Fill(MyDataSet, "LastName");
            return MyDataSet;
        }
    }
}
```

Так как класс `Service1` наследуется от `WebService`, то используется такая строка:

```
public class Service1 : System.Web.Services.WebService
```

Два метода, которые описаны с помощью атрибутов `WebMethod`, являются открытыми. Для обозначения открытых методов в языке C# используются квадратные скобки `[]`.

Примечание

В языке Visual Basic в данном случае используются символы <>.

Таким образом, открытые методы могут быть вызваны любым Web-клиентом.

Каждый открытый метод может иметь атрибуты (в примере из листинга 15.9 имеется только один атрибут `Description`). Эти атрибуты и их краткое пояснение приведено далее:

- `BufferResponse` — определяет, будет ли использоваться буфер для отклика метода. Может принимать значение `true` или `false`;
- `CacheDuration` — определяет время (в секундах), в течение которого отклик метода будет сохраняться в кэше. По умолчанию, значение равно `0`. То есть отклик метода в кэше не сохраняется;
- `Description` — включает описание данного Web-метода;

- `EnamleSession` — определяет, будет ли сохраняться состояние сеанса для Web-метода. Отключение состояния сеанса может поднять производительность. Принимает значение `true` или `false`. По умолчанию установлен в `true`;
- `MessageName` — используется в том случае, когда есть несколько методов с одинаковыми именами. Устанавливает отличия между этими методами. Если используется протокол SOAP, то данному свойству присваивается значение заголовка запроса `SOAPAction`, который располагается внутри элемента `<soap:Body>`. Если применяются протоколы HTTP GET или HTTP POST, то этому свойству присваивается значение, заданное в разделе `PathInfo` в URI;
- `TransactionOption` — определяет режим транзакции. Может принимать одно из следующих значений: `Disabled`, `NotSupported`, `Required`, `RequiresNew` и `Supported`. По умолчанию установлено значение `Disabled`.

Кроме asmx-файла, приведенного в листинге 15.9, для создания Web-службы необходимо предоставить конфигурационный файл Web.config, который должен располагаться в той же папке, что и файл asmx.

Файл Web.config позволяет управлять различными настройками приложения, относящимися к виртуальному каталогу. Главный элемент этого файла `<authentication mode>`, который задает режим аутентификации. Пока этот режим установлен в `None` — для облегчения разработки Web-службы.

Текст файла Web.config приведен в листинге 15.10.

Листинг 15.10. Файл Web.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

 <system.web>

  <!-- DYNAMIC DEBUG COMPILATION
       Set compilation debug="true" to enable ASPX debugging.
       Otherwise, setting this value to false will improve runtime
       performance of this application.
       Set compilation debug="true" to insert debugging symbols (.pdb
       information) into the compiled page. Because this creates a
       larger file that executes more slowly, you should set
       this_value to true only when debugging and to false at all
       other times. For more information, refer to the documentation
       about debugging ASP .NET files.
  -->
  <compilation
       defaultLanguage="c#"
```

```
      debug="true"
  />
  <!-- CUSTOM ERROR MESSAGES
      Set customError mode values to control the display of user-
      friendly error messages to users instead of error details
      (including a stack trace):
      "On" Always display custom (friendly) messages
      "Off" Always display detailed ASP.NET error information.
      "RemoteOnly" Display custom (friendly) messages only to users
      not running on the local Web server. This setting is
      recommended for security purposes, so that you do not display
      application detail information to remote clients.
  -->
  <customErrors
  mode="RemoteOnly"
  />

  <!-- AUTHENTICATION
      This section sets the authentication policies of the
      application. Possible modes are "Windows", "Forms",
      "Passport" and "None"
  -->
  <authentication mode="None" />

  <!-- APPLICATION-LEVEL TRACE LOGGING
      Application-level tracing enables trace log output for every
      page within an application.
      Set trace enabled="true" to enable application trace logging.
      If pageOutput="true", the trace information will be displayed
      at the bottom of each page. Otherwise, you can view the
      application trace log by browsing the "trace.axd" page from
      your web application root.
  -->
  <trace
      enabled="false"
      requestLimit="10"
      pageOutput="false"
      traceMode="SortByTime"
      localOnly="true"
  />

  <!-- SESSION STATE SETTINGS
      By default ASP .NET uses cookies—to identify which requests
      belong to a particular session.
```

```
      If cookies are not available, a session can be tracked by
      adding a session identifier to the URL.
      To disable cookies, set sessionState cookieless="true".
  -->
  <sessionState
      mode="InProc"
      stateConnectionString="tcpip=127.0.0.1:42424"
      sqlConnectionString="data source=127.0.0.1;
      user id=sa;password="
      cookieless="false"
      timeout="20"
  />

  <!-- GLOBALIZATION
      This section sets the globalization settings of the
      application.
  -->
  <globalization
      requestEncoding="utf-8"
      responseEncoding="utf-8"
  />

 </system.web>

</configuration>
```

Возможны следующие режимы аутентификации:

- `Forms` — при таком режиме запросы, не прошедшие аутентификацию, будут перенаправлены на форму регистрации;
- `None` — аутентификация не производится;
- `Passport` — запросы, не прошедшие аутентификацию, перенаправляются на централизованный сервис аутентификации Microsoft. После совершения аутентификации возвращается маркер, который используется в следующих запросах;
- `Windows` — аутентификация производится с помощью IIS.

Наконец, после создания Web-службы можно создать вспомогательные статические disco-файлы, для облегчения открытия Web-службы. Пример такого файла приведен в листинге 15.11.

Листинг 15.11. Статический disco-файл для Web-службы

```
<?xml version="1.0" encoding="utf-8" ?>
<disco:discovery xmlns:disco="http://schemas.xmlsoap.org/disco/"
```

```
xmlns:scl="http://schemas.xmlsoap.org/disco/scl/">
<scl:contractRef ref="http://localhost/MyWS/Service1.asmx?WSDL"/>
</disco:discovery>
```

Клиенты Web-служб

Клиенты работают с Web-службами с помощью стандартных Web-протоколов (HTTP GET, HTTP POST и SOAP). Они обмениваются с Web-службами сообщениями в формате XML. Таким образом, можно утверждать, что любое приложение на любой платформе может обратиться к сервисам Web-службы, если это приложение работает со стандартными Web-протоколами и может работать с языком XML.

Рассмотрим процесс создания клиентских приложений с помощью протоколов HTTP GET, HTTP POST и SOAP.

Web-клиент на основе HTTP GET

Для начала рассмотрим Web-клиента, использующего протокол HTTP GET, т. к. это самый простой протокол.

Если вы укажете Web-браузеру (например, Internet Explorer) в адресной строке URL Web-службы, например `http://localhost/MyWS/Service1.asmx` (листинг 15.11), то он отобразит список поддерживаемых Web-сервером методов.

Браузер автоматически сгенерирует простого клиента HTTP GET, который позволяет протестировать методы Web-службы.

Для вызова нужного метода Web-службы вы можете указывать в браузере полный URL, например:

```
http://localhost/MyWS/Service1.asmx?MyStr=Иванов
```

Web-клиент на основе HTTP POST и SOAP

Рассмотрим теперь, как Web-клиент может использовать протоколы HTTP GET или SOAP для доступа к Web-службе. Мы попробуем сами написать клиентское приложение на языке C#.

Для создания Web-служб и Web-клиентов среда .NET предоставляет специальный набор средств и утилит. Воспользуемся утилитой wsdl.exe, которая генерирует код "посредника" для уже существующих Web-служб. Эта утилита работает подобно компилятору IDL, который генерирует исходные тексты "посредников" для DCOM.

Примечание

WSDL — это основанный на XML язык, который используется для описания интерфейса компонента программного обеспечения.

Утилита wsdl.exe находится по умолчанию в папке: **Microsoft Visual Studio .NET 2003 | SDK | v1.1 | Bin**.

Сгенерируем "посредник" для Web-службы `MyWS` с помощью данной утилиты из командной строки:

```
wsdl/l:CS/protocol:HttpPost http://localhost/MyWS/Service1.asmx?WSDL
```

"Посредник" будет сгенерирован в виде исходного файла на языке C# и будет использовать протокол HTTP POST для работы с Web-службой. Вы могли бы указать в качестве параметра `protocol` значение `SOAP`:

```
wsdl/l:CS/protocol:SOAP http://localhost/MyWS/Service1.asmx?WSDL
```

В этом случае будет использоваться протокол SOAP.

После успешного создания исходного файла Service1.cs можно его использовать по-разному:

- включить исходный файл в проект клиентского приложения с помощью среды Visual Studio .NET. В этом случае, основной проект будет создаваться на языке C#;
- откомпилировать исходный файл в библиотеку DLL и добавить ссылку на нее в основной проект. Преимущество такого подхода в том, что проект может создаваться на любом из языков .NET.

Вы можете воспользоваться каким угодно способом. В любом случае код основного Web-клиента будет выглядеть абсолютно одинаково. Текст примера Web-клиента приведен в листинге 15.12.

Листинг 15.12. Web-клиент службы `MyWS`

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace MyTest1
{
   ///  <summary>
   ///  Summary description for Form1.
   ///  </summary>
```

```
public class Form1 : System.Windows.Forms.Form
{
   ///  <summary>
   ///  Required designer variable.
   ///  </summary>
   private System.ComponentModel.Container components = null;

   public Form1()
   {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      //
      //  TODO: Add any constructor code after
      //  InitializeComponent call
      //
   }

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

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

```
            this.Text = "Form1";
        }
        #endregion

        ///  <summary>
        ///  The main entry point for the application.
        ///  </summary>
        [STAThread]
        static void Main()
        {
            //  создание "посредника"
            MyWS MyProxy = new MyWS();

            //  вызов метода GetClients()
            DataSet MyDataSet = MyProxy.GetClients();

            //  создание сетки Data Grid и связывание с набором данных
            DataGrid MyDG = new DataGrid();
            MyDG.Size = new Size(300,300);
            MyDG.DataSource = MyDataSet.Tables["Customers"].DefaultView;

            //  отображение формы
            Application.Run(new Form1());
        }
    }
}
```

Если вы еще не создали DLL, то это можно сделать сейчас, с помощью утилиты csc.exe, которая по умолчанию устанавливается в папку:

Windows | Microsoft.NET | Framework | v1.0.3705

Для этого запишем такую командную строку:

```
Csc /t:library /r:system.web.services.dll /r:system.xml.dll
    /r:system.data.dll Service1.cs
```

Если вы уже создали DLL, то можно откомпилировать код, приведенный в листинге 15.12, с помощью той же утилиты csc.exe таким образом:

```
Csc MyTest1.cs /r:Service1.dll
```

Тем самым, мы создали исполняемый файл MyTest1.exe, который будет получать набор данных по протоколу HTTP POST и отображать сетку `Data Grid` с этим набором данных.

Безопасность Web-служб

Безопасность Web-служб является достаточно серьезным вопросом, который трудно описать в двух словах. Так как эта книга не предполагает раскрытие всех аспектов этой проблемы, кратко рассмотрим два способа защиты Web-служб:

1. *Безопасность на уровне системы*. Позволяет ограничивать доступ не зарегистрированных в системе клиентов. Осуществляется декларативным способом, путем задания IP-адресов зарегистрированных клиентов, которым разрешено пользоваться Web-службами.
2. *Безопасность на уровне приложения*. Авторизация пользователя происходит в самой Web-службе. Этим обеспечивается бóльшая гибкость, чем в первом способе.

Безопасность на уровне системы

Так как взаимодействие Web-службы с Web-клиентом осуществляется по протоколу HTTP, к ним можно применять безопасность на уровне системы точно так же, как и для обычных Web-страниц и ресурсов сайта.

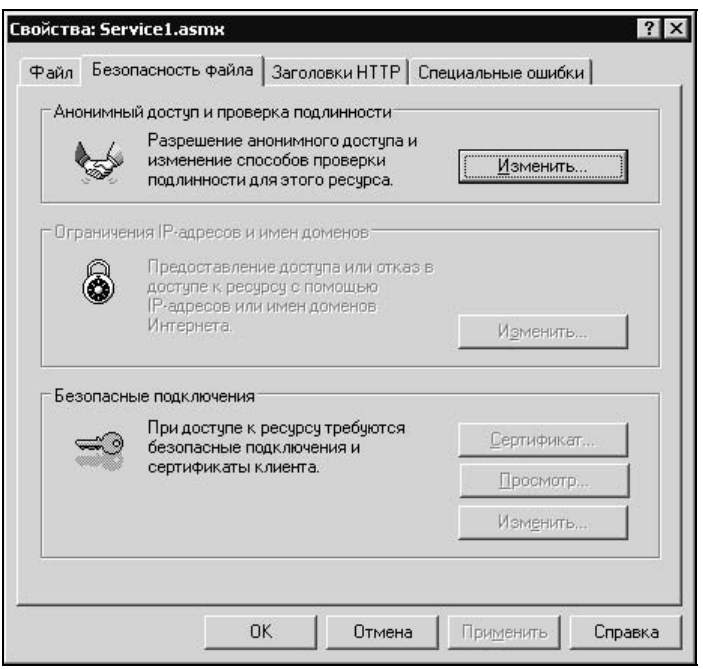


Рис. 15.2. Диалоговое окно свойств файла Service1.asmx, вкладка **Безопасность файла**

Защиту Web-служб можно осуществлять несколькими способами. Например, можно использовать утилиту администрирования IIS, которая может запрещать или разрешать доступ к Web-службе определенному набору IP-адресов, с помощью Internet Protocol Security (IPSec). Чтобы обезопасить себя от подмены IP-адресов, вы можете использовать брандмауэры.

Для защиты нашего примера Web-службы `MyWS` (см. листинг 15.2) запустите утилиту администрирования IIS и отредактируйте свойства для файла Service1.asmx в открывшемся диалоговом окне (рис. 15.2).

Нажмите кнопку **Изменить** в области **Анонимный доступ и проверка подлинности** на вкладке **Безопасность файла**. В появившемся диалоговом окне снимите флажок **Анонимный доступ**, обеспечивающий анонимный доступ к Web-службе (рис. 15.3).

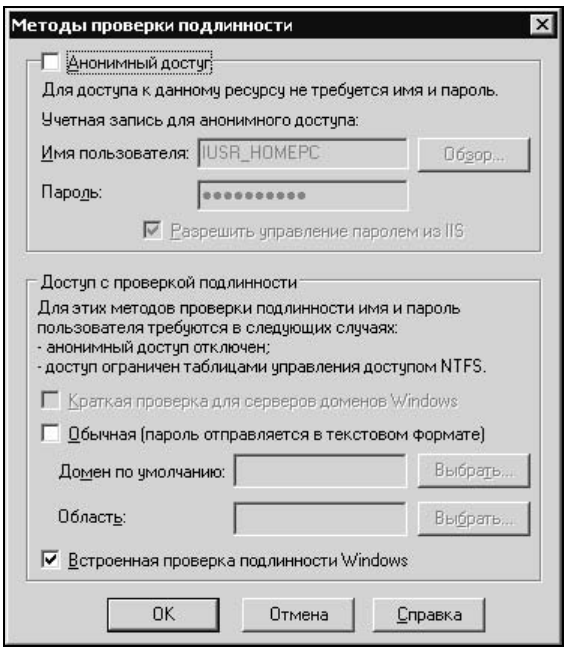


Рис. 15.3. Установка методов проверки подлинности

Безопасность на уровне приложений

Вы можете создать в Web-службах такой уровень безопасности, что для каждого метода будет требоваться маркер доступа, который клиент может получить от Web-службы путем отправки своего имени и пароля. Эти сведения могут отправляться на Web-сервер с использованием SSL-шифрования. По этому же SSL-каналу сервер может отправить маркер доступа.

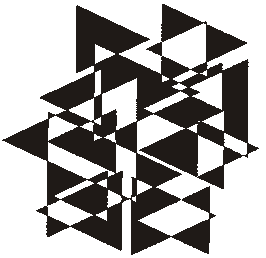
Единственный метод, который должен в данном случае работать на канале SSL, — это метод проверки имени пользователя и пароля `Login()`. После получения маркера доступа от сервера этот маркер можно использовать для вызова других методов Web-службы. Естественно, для избежания перехвата маркера "хакерами" необходимо убедиться, что IP клиента, использующего данный маркер, соответствует IP клиента, получившего его. Кроме того, вы можете ограничить использование полученного маркера по времени.

Если вы будете использовать такой способ обеспечения безопасности Web-служб, вам пригодятся службы Microsoft .NET Cryptographic Services. Эти службы поддерживают алгоритмы шифрования DES, RC2, RSA и TripleDES. Кроме того, они поддерживают методы "хэширования" SHA.

Все это позволит вам быстро создавать безопасные Web-службы.

Итак, мы рассмотрели основные принципы создания Web-служб и Web-клиентов в среде Visual Studio .NET 2003, а также создали собственных Web-службу и Web-клиента на языке C#. В *гл. 16* рассматривается технология ASP.NET, которая предназначена для создания динамических и интерактивных Web-страниц.

ГЛАВА 16



Создание Web-форм

В этой главе мы рассмотрим технологии, позволяющие создавать динамические Web-страницы. К этим технологиям относятся ASP и ASP.NET. В начале главы изучим проблемы, возникающие при создании Web-приложения, с использованием технологии ASP, а затем рассмотрим технологию ASP.NET и ее преимущества. Кроме того, в данной главе приводятся основные классы ASP.NET и синтаксис ASP.NET Web Forms. В конце главы показаны особенности создания Web-служб ASP.NET.

ASP и ASP.NET

Рассмотрим достоинства и недостатки технологии ASP и ее новой версии ASP.NET.

Технология ASP

Технология ASP (Active Server Pages) представляет собой технологию сценариев на стороне сервера, которая позволяет создавать динамические Web-страницы.

Данная технология появилась в 1997 году и предназначалась для создания Web-страниц в Internet Information Server.

Каждая страница ASP содержит обычную HTML-разметку и *серверные сценарии*, которые динамически генерируют HTML-содержимое.

Серверные сценарии активизируются, когда на сервер поступает запрос от клиента на получение ASP-страницы. Эти запросы формируются клиентскими браузерами с помощью методов HTTP GET и HTTP POST.

При всех достоинствах технология ASP имеет следующие недостатки:

- не очень хорошая модель инкапсуляции ASP-страниц. В результате, конфликты HTML-разметки и серверных сценариев;

- проблемы при генерации HTML, несоответствие возможностям браузеров клиентов. В итоге, вынужденная необходимость ограничиваться простейшими тегами HTML и клиентскими сценариями, а также отказ от расширенных возможностей хороших браузеров;
- нет безопасности типов в скриптах ASP, т. к. они доступны только на языках позднего связывания (например, JavaScript и VBScript);
- низкая производительность, т. к. серверные сценарии ASP-страниц интерпретируются при каждом обращении к странице;
- большие затраты времени и сил разработчика для поддержки состояния форм в ASP-приложении, т. к. практически все приходится делать вручную (повторный ввод данных, использование переменных сеанса и т. д.).

Таким образом, в технологии ASP имеется множество недостатков, большинство из которых были исправлены в усовершенствованной технологии ASP.NET.

Технология ASP.NET

Во многих средах быстрой разработки приложений используется так называемый визуальный метод создания форм. Примерами таких сред разработки являются Delphi, Visual Basic и многие другие. Создание форм в таких средах заключается в "перетаскивании" необходимых элементов управления на заготовку формы и написании обработчиков событий для этих элементов управления. Этот очень удобный подход к созданию форм используется и в ASP.NET.

Формы в ASP.NET носят название *Web-форм* (Web Forms). Web-формы аналогичны обычным формам в Visual Basic и, кроме того, Web-формы являются ASP-страницами.

Программирование Web-форм основано на событиях, в отличие от линейной программы ASP-страницы. Вам не нужно писать встраиваемые в ASP-страницу сценарии.

Преимущества технологии ASP.NET, по сравнению с более старой версией ASP, следующие:

- четкое разделение серверного кода и HTML-разметки;
- готовый набор серверных элементов управления, которые автоматически генерируют HTML-код для любых клиентов и управляют собственным состоянием;
- использование событийной модели программирования на сервере;
- расширенное управление состоянием сеанса;
- возможность написания приложения на любом .NET-языке;

- компиляция серверного кода приложения, что улучшает производительность;
- использование среды разработки Visual Studio .NET, позволяющей упростить процесс создания Web-форм.

Основные пространства имен и классы ASP.NET

Перед созданием первого приложения с использованием ASP.NET рассмотрим основные пространства имен, имеющих отношение к ASP.NET, и изучим важнейшие классы этих пространств имен.

Рассмотрим следующие пространства имен:

- `System.Web.UI`;
- `System.Web.UI.HtmlControls`;
- `System.Web.UI.WebControls`.

Пространство имен *System.Web.UI*

В данном пространстве имен определены классы и интерфейсы, которые используются при создании и отображении элементов управления Web-формы. Самым важным классом в этом пространстве имен является класс `Control`. Кроме этого класса к числу значимых классов можно отнести классы `Page` и `UserControl`.

Класс *Control*

Данный класс является базовым для всех элементов управления Web-форм. К элементам управления относятся такие элементы форм, как кнопки, текстовые поля, списки, раскрывающиеся списки и др. Таким образом, класс `Control` инкапсулирует в себе общую функциональность и свойства всех элементов управления.

Рассмотрим основные свойства и методы класса `Control`. Важнейшими свойствами класса являются:

- `ClientID` — позволяет получать идентификатор (ID) элемента управления, который был автоматически сгенерирован ASP.NET. ASP.NET всегда присваивает ID элементам управления, независимо от того, было ли установлено значение в свойстве `ID`;
- `Context` — позволяет получить информацию о текущем HTTP-запросе;
- `Controls` — позволяет выбрать объект `ControlCollection`, который представляет дочерние элементы указанного экземпляра элемента управления;

- `EnableViewState` — определяет, будет ли элемент управления сохранять свое визуальное состояние, а также визуальные состояния своих дочерних элементов. Например, если свойство `EnableViewState` имеет значение `true`, то предыдущий выбор пользователя или данные формы будут автоматически сохранены. И если пользователь покидает эту страницу, а затем возвращается к ней, он может продолжить заполнение формы, а все ранее введенные данные будут восстановлены;
- `ID` — позволяет получить или установить идентификатор (ID), который будет связан с данным элементом управления;
- `NamingContainer` — позволяет получить ссылку на пространство имен, в котором находится данный элемент управления. Пространство имен автоматически создается для того, чтобы была возможность отличить один элемент управления от другого, если они имеют одинаковые значения свойства `ID`;
- `Page` — позволяет получить ссылку на экземпляр класса `Page`, который содержит данный элемент управления (т. е. на Web-страницу, включающую данный элемент управления);
- `Parent` — позволяет получить ссылку на родительский элемент управления для данного элемента управления в пределах иерархии элементов управления Web-страницы;
- `Site` — позволяет получить информацию о сайте, на котором расположена страница, содержащая данный элемент управления;
- `ViewState` — представляет собой экземпляр класса `StateBag`, который служит для хранения информации в виде пар имя-значение, позволяющий обрабатывать несколько запросов к одной Web-странице;
- `Visible` — определяет, будет ли отображаться элемент управления на Web-форме.

Далее приведены основные методы класса `Control`:

- `DataBind` — связывает элемент управления с источником данных. Метод вызывается в сочетании с синтаксисом выражений привязки к данным в Web-форме. При вызове метода все теги привязки к данным (`<%# %>`) вычисляются заново, для того чтобы в местах расположения тегов находились новые данные;
- `CreateChildControls` — этот метод вызывается непосредственно перед отображением любого составного нестандартного элемента управления. Такой элемент управления аналогичен элементу управления ActiveX, т. к. составлен из других элементов управления;
- `LoadViewState` — загружает состояние элемента управления;
- `Render` — в основном применяется для разработки нестандартных элементов управления. Разработчики элементов управления могут выклю-

чить данный метод для формирования содержимого элемента управления через параметр `HtmlTextWriter`;

- `SaveViewState` — сохраняет состояние элемента управления (например, в промежутках между запросами).

Класс *Page*

Данный класс является потомком класса `Control`. Таким образом, он наследует все свойства, события и методы, которые есть у класса `Control`. Все Web-страницы ASP.NET представляют собой экземпляры классов, предком которых является класс `Page`.

Перечислим основные свойства класса `Page`:

- `Application` — позволяет обращаться к объекту `Application` текущего Web-запроса;
- `Cache` — позволяет обращаться к объекту `Cache` текущей Web-страницы. В данном объекте сохраняются для повторного использования различные ресурсы;
- `ErrorPage` — позволяет установить Web-страницу, которая будет отображаться в случае возникновения ошибки. Заметим, что данную страницу можно установить и с помощью тега `<%@Page>`. Далее приведен пример установки страницы с сообщением об ошибке (на языке C#):

```
void Page_Load(Object sender, EventArgs e)
{
   if(!IsPostBack)
   this.ErrorPage = "Error_Page.aspx";
}
```

- `IsPostBack` — определяет, будет ли выполняться запрос к странице как в первый раз, или происходить возврат к ней. Если значение данного свойства `true`, то повторную инициализацию страницы выполнять не требуется, что положительно сказывается на производительности;
- `Request` — позволяет обращаться к объекту `HttpRequest` запрашиваемой страницы. Данный объект содержит информацию о текущем HTTP-запросе;
- `Response` — позволяет выбрать объект `HttpResponse`, связанный с данной Web-страницей. Этот объект передает данные HTTP-ответа сервера клиенту и содержит информацию об этом ответе;
- `Server` — позволяет выбрать объект `Server`, связанный с данной страницей;
- `Session` — позволяет получить данные о текущем состоянии сессии;

- `Trace` — позволяет обращаться к объекту `TraceContext` для текущего Web-запроса. Данный объект выдает предупреждения и сообщения об ошибках. Такая трассировка может быть в любое время выключена или вновь включена с помощью настроек в текстовом XML-файле Web.config. Изменения в данном файле вступают в силу немедленно;
- `Validators` — позволяет получить коллекцию серверных элементов управления, которые могут подтверждать свою правильность.

Перечислим основные методы класса `Page`:

- `CreateHtmlTextWriter` — создает объект `HtmlTextWriter` для выдачи HTML-кода в выходной поток данных. Работает по аналогии с методом `Response.Write` в ASP, но является более правильным и корректным;
- `LoadControl` — программным методом загружает элементы управления на Web-страницу;
- `LoadPageStateToPersistenceMedium` — загружает визуальное состояние всех элементов управления в виде скрытых полей Web-страницы;
- `MapPath` — устанавливает соответствие между физическим путем и виртуальным для файлового ввода/вывода. Данный метод должен быть вам хорошо знаком, если вы уже создавали приложения с использованием ASP;
- `SavePageStateToPersistenceMedium` — сохраняет визуальное состояние всех элементов управления в виде скрытых полей Web-страницы;
- `Validate` — работает с проверяющими правильность элементами управления для проверки достоверности данных на странице. Если какой-либо элемент управления не прошел проверку, то данный метод возвратит значение `false`, кроме того, элемент управления выдаст пользователю сообщение об ошибке.

Класс *UserControl*

Данный класс похож на класс `Page`, за исключением свойств и методов, которые относятся к Web-страницам (`ErrorPage`, `Validators`, `MapPath`, `Validate` и др.).

Класс `UserControl` применяется в качестве базового класса для нестандартных элементов управления. Хотя, вы можете использовать в качестве базового класса для нестандартных элементов управления и класс `Control`.

Пространство имен *System.Web.UI.HtmlControls*

Технология ASP.NET ставит в соответствие тегам HTML объекты иерархии серверных классов, которые определены в пространстве имен `System.Web.UI.HtmlControls`. Такие серверные объекты носят название `HtmlControls`.

В листинге 16.1 приведен текст простой HTML-страницы, которая использует клиентские сценарии для своего динамического изменения.

Листинг 16.1. Простая динамическая HTML-страница

```
<html>
<head>
   <script language=vbscript>
   sub cmd1_onclick()
   txtMessage.InnerHtml = _
   "Введен текст: " & frm1.txtName.value
   end sub
   </script>
</head>
<body>
   <form id=frm1>
   Введите произвольный текст: <input id="txtName" type="text" size="40">
   <input type=button id="cmd1" value="OK">
   <span id="txtMessage"></span>
   </form>
</body>
</html>
```

Примечание

HTML-страница (листинг 16.1) не будет работать в Web-браузерах, которые не поддерживают клиентские сценарии VBScript или в которых поддержка этих сценариев отключена.

Если загрузить HTML-страницу с помощью браузера и ввести в поле ввода произвольный текст, а затем нажать кнопку **OK**, данный текст отобразится справа от кнопки (рис. 16.1).

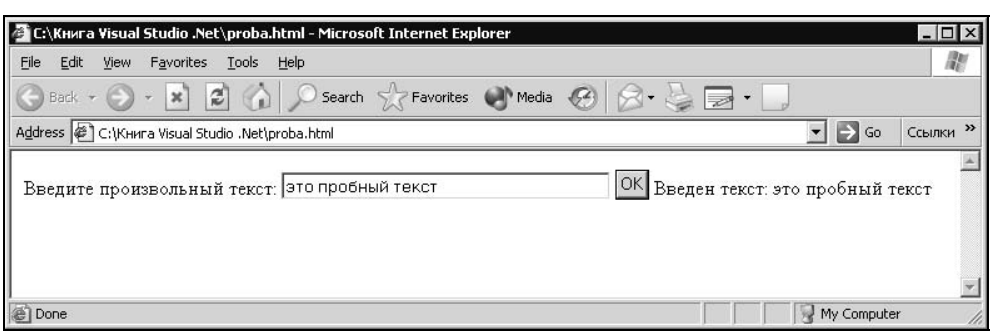


Рис. 16.1. Работающая динамическая HTML-страница

Преобразуем HTML-страницу таким образом, чтобы она была основана на серверных элементах управления, а не на объектной модели документа Internet Explorer. Вывод страницы будет осуществляться со стороны сервера, поэтому неважно, с помощью какого браузера вы будете ее просматривать.

Все, что необходимо сделать, — это добавить атрибуты `id` и `runat`. Текст HTML-страницы с данными атрибутами приведен в листинге 16.2.

Листинг 16.2. HTML-страница, основанная на серверных элементах управления

```
<html>
<head>
   <script id="scr1" language="c#" runat="server">
   void svr_cmd1_onclick(Object o, eventArgs e)
   {
      txtMessage.InnerHtml =
      "Введен текст: " + txtName.Value
   }
   </script>
</head>
<body>
   <form id=frm1 runat="server">
   Введите произвольный текст: <input id="txtName"
   type="text" size="40" runat="server">
   <input type=button id="cmd1" value="OK">
   <span id="txtMessage" runat="server"></span>
   </form>
</body>
</html>
```

Атрибут `runat="server"` позволяет использовать серверным сценарием объекты `HtmlForm`, `HtmlInputText`, `HtmlInputButton` и `HtmlGenericControl`.

Результаты работы примеров из листингов 16.1 и 16.2 аналогичны. Но существует серьезное различие: пример из листинга 16.1 работает в рамках процесса клиентского браузера, а пример из листинга 16.2 — на сервере.

Иерархия классов пространства имен `System.Web.UI.HtmlControls` представлена на рис. 16.2.

Базовым классом пространства имен `System.Web.UI.HtmlControls` является класс `HtmlControl`, который, в свою очередь, является потомком класса `Control`.

Так как каждому тегу HTML соответствует один класс `HtmlControl`, приведу список соответствий между ними (табл. 16.1).

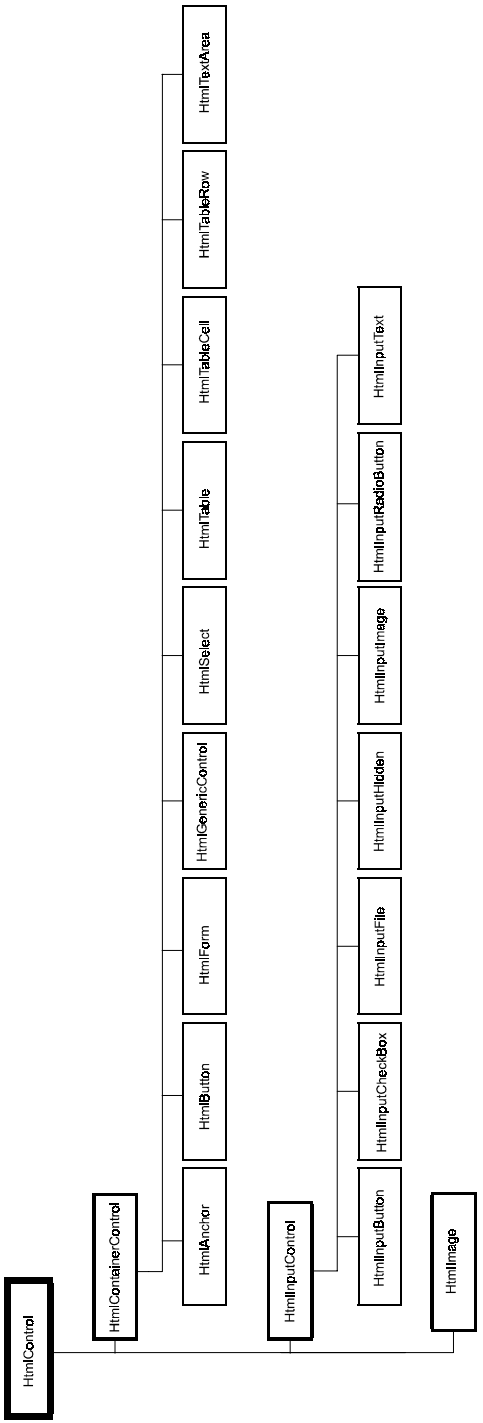


Рис. 16.2. Иерархия классов `HtmlControl`

Таблица 16.1. *Соответствия между тегами HTML и объектами* `HtmlControls`

Теги HTML	Объект `HtmlControl`	Краткое описание
`<img>`	`HtmlImage`	Графическое изображение
`<input type="file">`	`HtmlInputFile`	Выбор файла
`<input type="hidden">`	`HtmlInputHidden`	Скрытое поле формы
`<input type="image">`	`HtmlInputImage`	Ввод изображения
`<input type="radio">`	`HtmlInputRadioButton`	Переключатель
`<input type="text">`	`HtmlInputText`	Текстовое поле для ввода
`<input type="button">`	`HtmlInputButton`	Кнопка HTML
`<input type="checkbox">`	`HtmlInputCheckBox`	Флажок HTML
`<form>`	`HtmlForm`	Форма
`<div>`, `<span>` и др.	`HtmlGenericControl`	Группирующие теги HTML
`<select>`	`HtmlSelect`	Поле с раскрывающимся списком
`<table>`	`HtmlTable`	Таблица HTML
`<td>`	`HtmlTableCell`	Ячейка таблицы
`<tr>`	`HtmlTableRow`	Строка таблицы
`<textarea rows=n cols=k>`	`HtmlTextArea`	Многострочное текстовое поле для ввода
`<a href= >` или `<a name= >`	`HtmlAnchor`	Гиперссылка HTML

Таким образом, если имеется тег HTML, помеченный как работающий на стороне сервера (`runat="server"`), то ASP.NET создаст соответствующий объект `HtmlControl`, который затем можно использовать в сценарии.

Пространство имен *System.Web.UI.WebControls*

Кроме уже рассмотренных объектов `HTMLControls`, в ASP.NET имеются дополнительные объекты, позволяющие создавать другие элементы интерфейса пользователя. К таким элементам можно отнести таблицы, календари и др.

Все эти объекты находятся в пространстве имен `System.Web.UI.WebControls` и являются более мощными по сравнению с объектами `HtmlControls`. Но

объекты `HtmlControls` больше подходят для переноса старых приложений ASP на новую технологию ASP.NET.

Базовым классом пространства имен `System.Web.UI.WebControls` считается класс `WebControl`, являющийся потомком класса `Control`.

Приведу иерархию классов пространства имен `System.Web.UI.WebControls`:

```
WebControl
        ListControl
            RadioButtonList
            CheckBoxList
            DropDownList
            ListBox
        BaseDataList
            DataList
            DataGrid
        Label
            BaseValidator
                CustomValidator
                RegularExpressionValidator
                RequiredFieldValidator
                BaseCompareValidator
                      CompareValidator
                      RangeValidator
        ValidationSummary
        AdRotator
        Button
        Calendar
        CheckBox
            RadioButton
        DataListItem
        HyperLink
        Image
            ImageButton
        LinkButton
        Panel
        TextBox
        Table
        TableCell
            TableHeaderCell
        TableRow
            DataGridItem
```

Синтаксис Web-форм

Web-формы представляют собой текстовые файлы, которые состоят из тегов HTML и управляющих тегов (директив и блоков сценариев). В этом Web-формы похожи на файлы ASP. Файл Web-формы обычно имеет расширение aspx.

Рассмотрим некоторые элементы синтаксиса, которые используются в ASP.NET:

- директивы;
- блоки объявления кода;
- синтаксис элементов управления HTML;
- синтаксис нестандартных элементов управления;
- выражения для привязки данных;
- теги серверных объектов.

Директивы

В ASP все директивы имели следующий синтаксис:

```
<%@ [атрибут=значение], [атрибут=значение], ... %>
```

В ASP.NET синтаксис директив изменился, т. к. появилось несколько различных директив (`Assembly`, `Control`, `Implements`, `Import`, `OutputCache`, `Page`, `Register` и `Reference`). Теперь все директивы ASP являются атрибутами директивы `Page`, а общий синтаксис для всех директив принял следующий вид:

```
<%@ directive [атрибут=значение], [атрибут=значение], ... %>
```

Рассмотрим вышеупомянутые директивы:

- `Assembly` — применяется для указания на сборку, к которой принадлежит текущая страница. Использование директивы делает доступными все классы и интерфейсы, принадлежащие к данной сборке на текущей странице. Например, запись `<%@ Assembly Name="Ch02">` означает, что текущая страница относится к сборке `Ch02` и, соответственно, в коде данной страницы допустимо обращаться ко всем классам и интерфейсам данной сборки;
- `Control` — используется для элемента управления ASP.NET (файл с расширением ascx);
- `Implements` — позволяет объявить, что aspx-файл реализует какой-либо интерфейс. Например, следующая запись: `<%@ Implements Interface ="System.Web.UI.IPostBackEventHandler" %>` означает, что HTML-страница реализует интерфейс `System.Web.UI.IPostBackEventHandler`;

- `Import` — применяется для добавления в текущую страницу ссылок на используемые пространства имен. Указав эту директиву, вы можете использовать все классы и интерфейсы из импортированных пространств имен. Например, запись вида: `<%@ Import Namespace="System.NET" %>` позволяет использовать все базовые классы .NET Framework из пространства имен `System.NET` на текущей странице;
- `OutputCache` — используется для управления временем выходного кэширования текущей страницы. Работает по аналогии с установкой времени истечения срока действия в объекте `Response` при работе с ASP. Атрибут `Duration` позволяет установить время действия страницы в секундах: `<%@ OutputCache Duration="10" %>`;
- `Page` — включает в себя все ASP-директивы. Кроме того, поддерживает такие атрибуты, как `EnableViewState`, `ErrorPage`, `Inherits`, `Src` и др. Запись с использованием директивы `Page` может быть такая: `<%@ Page Language="VB" ContentType="text/xml" %>`;
- `Register` — применяется для регистрации нестандартных серверных элементов управления с помощью задания псевдонимов для префиксов имен классов. Например: `<%@ Register Tagprefix="MyTag" TagName="MyNewControl", Src="MyNewControl.ascx" %>`. Таким образом, имя нового элемента управления `MyNewControl`, а префикс, задаваемый при объявлении элемента управления, — `MyTag`. Исходный текст элемента управления должен находиться в файле MyNewControl.ascx;
- `Reference` — применяется для добавления ссылки на страницу или элемент управления. Например: `<%@ Reference Control="MyControl.ascx" %>`. В данном случае aspx-страница будет ссылаться на элемент управления (файл) MyControl.ascx.

Блоки объявления кода

Блоки объявления кода в ASP.NET так же, как и в ASP, определяют код страницы, который должен быть запущен на выполнение.

Атрибут `runat` блока объявления кода указывает, как должен выполняться код: на сервере или клиентом. Отсутствие данного атрибута означает, что данный блок должен выполняться клиентом, а значение атрибута `runat`, равное `server`, показывает, что код должен быть выполнен сервером.

"Каркас" блока объявления кода, выполняемый сервером, выглядит так:

```
<script runat="server" language="язык_программирования">
   код
</script>
```

Пример блока объявления кода приведен в листинге 16.3. Этот пример может быть использован в приложении ASP.NET для определения четырех

обработчиков событий. Используемый язык программирования — Visual Basic .NET. Данный блок работает на уровне сервера.

Листинг 16.3. Блок объявления кода

```
<script language="VB" runat="server">
   Sub Application_OnStart()
      ' Здесь располагается код, выполняемый при запуске приложения.
   End Sub
   Sub Session_OnStart()
      ' Здесь располагается код, выполняемый при запуске сессии.
   End Sub
   Sub Session_OnEnd()
      ' Здесь располагается код, выполняемый при завершении сессии.
   End Sub
   Sub Application_OnEnd()
      ' Здесь располагается код, выполняемый при завершении приложения.
   End Sub
   Overrides Sub HandleError(ErrorInfo as Exception)
      ' Здесь располагается код, выполняемый при возникновении ошибок.
   End Sub
</script>
```

Кроме атрибута `runat`, вы можете задать атрибут `src`, который ссылается на внешний исходный файл, содержащий непосредственно код (который может быть как клиентским, так и серверным). Данный атрибут используется для разделения кода приложения и содержимого HTML-страницы.

В качестве значения атрибута `src` может выступать как URL файла, так и относительный путь.

Синтаксис элементов управления HTML

Все элементы управления HTML похожи на стандартные теги HTML. Исключение составляют только атрибуты `runat` и `id`, которые присутствуют в элементах управления HTML. Однако атрибут `id` должен быть знаком вам, если вы ранее создавали Web-приложения с помощью Dynamic HTML.

В качестве примера приведу код, который представляет собой элемент управления кнопку и должен выполняться на сервере:

```
<input id="cmd1" runat="server" type="button" value="OK" />
```

Данный код генерирует серверный объект `HtmlControl`, имеющий тип `HtmlInputButton` со значением `id="cmd1"`.

Примечание

Все серверные элементы управления HTML должны располагаться внутри элемента управления `form`, т. к. Web-формы применяют метод `Post` для сохранения состояний своих элементов управления.

Вы можете создать для кнопки обработчики событий, например обработчик события нажатия на кнопку. Для этого необходимо воспользоваться одним из следующих способов:

- указать внутри тега элемента управления имя события и присвоить ему значение, равное имени функции обработчика этого события. Например, используя такую запись: `<input id="cmd1" runat="server" type="button" value="OK", onserverclick="handleServerClick" />`;
- написать строку кода, которая присвоит делегат свойству события элемента управления. Например, таким образом (на языке C#): `cmd1.ServerClick += new System.EventHandler(handleServerClick)`.

Все обработчики событий обязательно должны иметь два параметра:

- `Sender` (тип `Object`) — указывает на исходный элемент управления, который вызвал событие;
- `Event` (тип `EventArgs`) — означает непосредственно произошедшее событие.

Синтаксис нестандартных элементов управления

Нестандартные элементы управления, как и стандартные, имеют атрибуты `runat` и `id`. Общий синтаксис нестандартного элемента управления следующий:

```
<tagprefix:tagname id="controlID" runat="server"
 eventname="eventHandler" />
```

Обратите внимание на тот факт, что все теги нестандартных элементов управления имеют префикс, являющийся псевдонимом пространства имен, в котором определен этот элемент управления.

Привязка событий к нестандартным элементам управления осуществляется так же, как и к стандартным элементам управления.

Выражения привязки данных

Выражения привязки данных позволяют связывать серверные элементы управления с источниками данных. Общий синтаксис привязки данных следующий:

```
<%# выражение привязки данных %>
```

В приведенном ниже примере (листинг 16.4) можно видеть привязку текстового содержимого тега `label` к переменной `MyText`.

Листинг 16.4. Пример привязки данных

```
<html>
   <head>
   <title>Привязка данных</title>
   </head>
   <body>
   <script language="C#" runat=server>
   /* Объявление текстовой переменной,
   к которой будет выполнена привязка */
   public string MyText;
   void Page_Load(Object oSender, EventArgs oEvent)
   {
      MyText="Это пример текста";
      Page.DataBind();
   }
   </script>

   <asp:Label text='<%# MyText %>' runat=server />

   </body>
</html>
```

Обратите внимание на тот факт, что привязка данных не может осуществляться автоматически. Вам нужно явно вызывать метод `DataBind()`.

Теги серверных объектов

Теги серверных объектов используются для объявления и создания .NET- и .COM-объектов. Они имеют следующий синтаксис:

```
<object id="id" runat="server" scope="scope" class="имя класса .NET">
<object id="id" runat="server" scope="scope" progid="COM_ProgID">
<object id="id" runat="server" scope="scope" classid="COM_classID">
```

Параметр `scope` (диапазон) указывает, в каком качестве будет доступен объект на данной странице (в качестве переменной приложения или переменной сеанса).

Для динамического добавления серверного объекта на страницу используйте метод `Page.LoadControl()`.

Разработка приложения ASP.NET

При создании обычных ASP-приложений для получения параметров, необходимых для отображения страниц, используется объект `Request`. Содержимое страницы выводится в блоках генерации кода или с помощью объекта `Response`. Для управления переменными приложения и настройками сервера, а также для других целей, в ASP-приложениях используются объекты `Application`, `Server`, `Session` и др.

В ASP.NET программисту не нужно возвращаться к старым методам. Рассмотрим методы создания приложений ASP.NET. Начнем с компонентов Web-формы.

Компоненты Web-формы

Web-формы состоят из двух частей:

- непосредственно формы, с расположенными на ней элементами управления;
- кода обработчиков событий, связанных с элементами управления формы.

Файл Web-формы имеет расширение aspx и содержит внутри себя HTML-элементы и серверные элементы управления. Код обработчика событий обычно располагается в отдельном файле класса, но допустимо размещать его и в одном файле формы.

Примечание

Я рекомендую вам располагать код обработчика событий и форму в разных файлах во избежание путаницы.

В aspx-файле задается схема страницы и производится объявление элементов управления.

Итак, для создания приложения ASP.NET вы должны предоставить *пользовательский интерфейс Web-формы* и *код, обслуживающий класс.*

Пользовательский интерфейс Web-формы — это объявление серверных элементов управления с соответствующими идентификаторами ID.

Код, обслуживающий класс, — это программа, которая работает с серверными элементами управления (они объявлены в пользовательском интерфейсе), а также обрабатывающая события данных элементов управления.

Рассмотрим простой пример aspx-файла (листинг 16.5) и его обслуживающий код (листинг 16.6).

Листинг 16.5. Aspx-файл

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="false" Inherits="MyWA1.WebForm1" %>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
   <head>
      <title>Пример работы обслуживающего кода</title>
      <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
      <meta name="CODE_LANGUAGE" Content="C#">
      <meta name="vs_defaultClientScript" content="JavaScript">
      <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
   </head>
   <body MS_POSITIONING="GridLayout">
       <form id="Form1" method="post" runat="server">
       Время инициализации: <asp:Label ID=MylabInit Runat=server /><br/>
       Время загрузки: <asp:Label ID=MylabLoad Runat=server /><br/>
       <input type=submit />
       </form>
   </body>
</html>
```

Данный aspx-файл содержит только теги HTML и директиву, которая ссылается на обслуживающий код.

Листинг 16.6. Обслуживающий код

```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace MyWA1
{
   ///  <summary>
   ///  Summary description for WebForm1.
   ///  </summary>
   public class WebForm1 : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Label MylabInit;
      protected System.Web.UI.WebControls.Label MylabLoad;

      public WebForm1()
      {
         MylabInit = new System.Web.UI.WebControls.Label();
         MylabLoad = new System.Web.UI.WebControls.Label();
      }
```

```
    public void Page_Init(object oSender, EventArgs oEvent)
    {
       MylabInit.Text = DateTime.Now.ToString();
    }
    public void Page_Load(object sender, System.EventArgs e)
    {
       MylabLoad.Text = DateTime.Now.ToString();
       if(IsPostBack)
       {
          MylabLoad.Text += "(PostBack)";
       }
    }

    #region Web Form Designer generated code
    override protected void OnInit(EventArgs e)
    {
       //
       //  CODEGEN: This call is required by the
       //  ASP.NET Web Form Designer.
       //
       InitializeComponent();
       base.OnInit(e);
    }

    ///  <summary>
    ///  Required method for Designer support — do not modify
    ///  the contents of this method with the code editor.
    ///  </summary>
    private void InitializeComponent()
    {
       this.Load += new System.EventHandler(this.Page_Load);
    }
    #endregion
  }
}
```

Данный обслуживающий код содержит класс `WebForm1`, на который ссылается aspx-файл.

Теперь необходимо откомпилировать обслуживающий код и поместить созданную библиотеку DLL в подкаталог **bin** виртуального каталога приложения (рис. 16.3). Только после этого можно отобразить aspx-страницу в Web-браузере.

Для создания виртуального каталога приложения вы можете воспользоваться утилитой администрирования IIS.

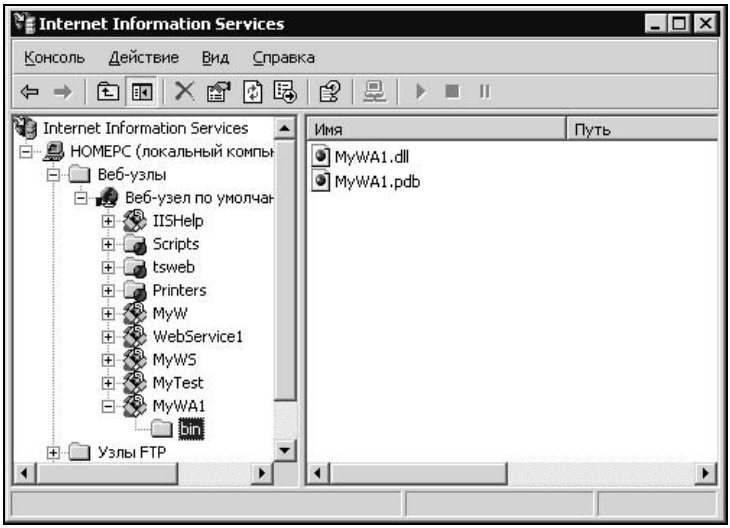


Рис. 16.3. Сохранение файла библиотеки

События Web-формы

В примере из листинга 16.6 использовались два события класса `Page`: `Init` и `Load`. Кроме этих событий класс `Page` предоставляет еще такие события, как `PreRender` и `Unload`.

Вы можете писать обработчики для этих событий, программируя работу своей Web-формы.

Первое событие, которое происходит с Web-формой, — это событие инициализации формы `Init`. На данном этапе элементы управления Web-формы еще не созданы и не отображаются. Это событие возникает один раз для каждого пользователя страницы.

Далее следует событие `Load`. Оно возникает всякий раз при запрашивании страницы клиентом. При возникновении данного события все элементы управления формы уже созданы и доступны.

Событие `PreRender` возникает непосредственно перед генерацией страницы и отправки ее клиенту.

И последнее событие, которое происходит с Web-формой, — это событие выгрузки формы из памяти `Unload`.

Серверные элементы управления

Серверные элементы управления представляют собой программируемые объекты, которые запускаются на сервере перед непосредственной генера-

цией Web-страницы средой ASP.NET. Элементы управления сохраняют свое состояние на сервере между запросами к странице, на которой они находятся, путем вставки скрытого поля с данными о визуальном состоянии формы. Этим самым исключается необходимость повторного заполнения ранее заполненных полей формы при возврате на страницу.

Серверные элементы управления, исходя из их названия, работают на стороне сервера, этим достигается независимость браузера, с помощью которого клиент обращается к Web-странице. Кроме того, серверные элементы управления могут применять свойство `Request.Browser`, которое содержит информацию об использованном клиентом браузере.

В общем смысле, серверные элементы управления представляют собой экземпляры классов .NET. Таким образом, программирование таких элементов управления приводит к созданию кода, который затем легко поддерживать и модифицировать.

Вместе со средой ASP.NET поставляются серверные элементы управления, которые мы уже рассматривали ранее при изучении пространств имен `System.Web.UI.HtmlControls` и `System.Web.UI.Webcontrols`.

Создание Web-служб с помощью ASP.NET

Использование ASP.NET сильно упрощает создание Web-служб, т. к. заботы по упаковке и распаковке данных в формат XML и использование протокола HTTP выполняется на уровне среды.

Для файлов Web-служб в среде .NET используется специальное расширение asmx.

Все файлы Web-служб должны начинаться с директивы `@WebService`, которая указывает ASP.NET на способ компиляции кода и задает имя основного класса. Данная директива имеет следующие атрибуты:

- `Class` — служит для определения класса, который предоставляет Web-методы. Среда ASP.NET создает экземпляр указанного класса для того, чтобы клиенты могли вызывать его методы;
- `Codebehind` — служит для разделения кода и ASP-страницы. Позволяет указать местонахождение и название файла, содержащего исходный код;
- `Language` — применяется для указания языка, на котором написан код. Вы можете указать один из трех языков: C# (`C#`), Visual Basic .NET (`VB`) или JScript .NET (`JS`).

В листинге 16.7 приведен пример простой Web-службы, созданной с помощью Visual Studio .NET.

Листинг 16.7. Web-служба

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace MyWebS1
{
   ///  <summary>
   ///  Summary description for Service1.
   ///  </summary>
   public class Service1 : System.Web.Services.WebService
   {
      public Service1()
      {
         //  CODEGEN: This call is required by the ASP.NET Web
         //  Services Designer
         InitializeComponent();
      }

      #region Component Designer generated code

      //  Required by the Web Services Designer
      private IContainer components = null;

      ///  <summary>
      ///  Required method for Designer support — do not modify
      ///  the contents of this method with the code editor.
      ///  </summary>
      private void InitializeComponent()
      {
      }

      ///  <summary>
      ///  Clean up any resources being used.
      ///  </summary>
      protected override void Dispose( bool disposing)
      {
         if(disposing && components != null)
         {
            components.Dispose();
         }
```

```
        base.Dispose(disposing);
    }

    #endregion

    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
  }
}
```

Для создания Web-службы выберите в области **Templates** диалогового окна **New Project** пиктограмму **ASP.NET Web Service**, как показано на рис. 16.4.

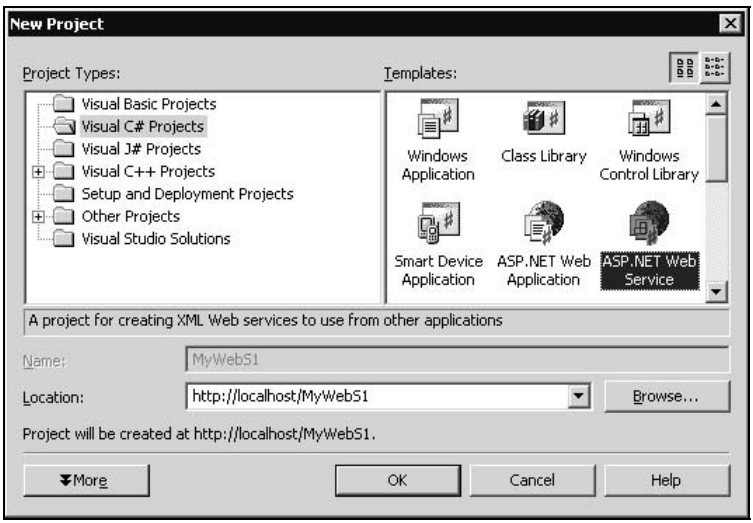


Рис. 16.4. Создание новой ASP.NET Web-службы

При просмотре содержимого сгенерированного средой .NET asmx-файла вы увидите следующую строку с директивой `@WebService`:

```
<%@ WebService Language="c#" Codebehind="Service1.asmx.cs"
Class="MyWebS1.Service1" %>
```

Таким образом, используемый язык — C#, код находится во внешнем файле Service1.asmx.cs, а класс, предоставляющий Web-методы — `Service1`.

Данная Web-служба способна возвращать строку `Hello World`.

Как вы уже видели в листинге 16.7, для того чтобы пометить открытые методы классов, используется специальный атрибут `WebMethod`. Помеченные

этим атрибутом методы становятся доступны через Интернет. В языке C# синтаксис атрибута имеет следующий вид:

```
[WebMethod(атрибут="значение" атрибут="значение" ...)]
public Тип_возвращаемого_значения Имя_функции(список_параметров)
```

Примечание

В разных языках синтаксис атрибутов метода отличается. Например, в Visual Basic вместо квадратных скобок используются угловые <>, а для присваивания используется знак := вместо =.

Привязка данных и применение шаблонов

Все элементы управления ASP.NET могут иметь привязку к данным, но только элементы управления `DataGrid`, `DataList` и `Repeater` используют шаблон для отображения данных.

Рассмотрим простейший пример формы с привязкой данных. В файле формы присутствуют только два списка с идентификаторами `listbox1` и `listbox2`:

```
<asp:ListBox ID="listbox1" Runat="server"></asp:ListBox>
<asp:ListBox ID="Listbox2" Runat="server"></asp:ListBox>
```

В обработчике события `Page_Load` в файле обслуживающего кода создается два источника данных `Array`, которые реализуют интерфейс `ICollection`. После чего выполняется привязка списков к этим источникам данных, с помощью метода `DataBind()` (листинг 16.8).

Листинг 16.8. Код, располагающийся внутри обработчика события `Page_Load`

```
int[] MyArray1 = new int[7] {1, 2, 3, 4, 5, 6, 7};
string[] MyArray2 = new string[7] {"Понедельник", "Вторник", "Среда",
                  "Четверг", "Пятница", "Суббота", "Воскресенье"};
listbox1.DataSource = MyArray1;
listbox1.DataBind();
listbox2.DataSource = MyArray2;
listbox2.DataBind();
```

Результат работы данного фрагмента представлен на рис. 16.5.

Рис. 16.5. Пример привязки данных из массивов

Управление состоянием сеансов

Состояние сеанса — это переменная, кэшируемая на сервере во время сеанса пользователя. Пока клиент работает с Web-приложением, состояние сеанса сохраняет свое значение вплоть до завершения сеанса.

В ASP управление состоянием сеанса осуществлялось таким образом:

1. В начале сеанса Web-приложение присваивает клиенту уникальный ключ.
2. Ключ, присвоенный пользователю, сохраняется в файлах Cookie. При последующих вызовах браузер клиента отправляет данный ключ обратно серверу.
3. Сервер обрабатывает полученный ключ и, в соответствии с ним, обрабатывает запрос клиента.

Эта технология хороша, но имеет одно большое ограничение: состояние сеанса зависит от процесса.

Технология ASP.NET позволяет перейти к внепроцессной модели. Все Web-серверы указывают на общий сервер, на котором работает внепроцессный диспетчер состояний. Таким образом, клиент может перенаправляться на различные серверы, не теряя своих состояний сеанса.

Внепроцессная модель избавляет от многих проблем. Например, если в приложении Web-сервера возник сбой, и оно было перезапущено во время открытых сеансов, все состояния Web-клиентов сохранятся. Естественно, состояния Web-клиентов не сохраняются, если сбой происходит во внепроцессном диспетчере состояний, но здесь ASP.NET предлагает возможность сохранять состояние сеансов в базе данных.

В табл. 16.2 приведены достоинства и недостатки двух моделей: внутрипроцессной (ASP) и внепроцессной (ASP.NET). Кроме того, приведена еще одна модель управления состоянием сеанса — с помощью SQL Server.

Таблица 16.2. *Описание режимов управления состоянием сеансов*

Режим управления состоянием сеансов	Краткое описание
Внутрипроцессный	Данный режим позволяет достичь наибольшей производительности, но наименьшей надежности (из-за сохранения в памяти). Режим не является масштабируемым, т. к. базируется на процессе. Программисту приходится самостоятельно обеспечивать, чтобы при последующих запросах клиент обращался к тому же серверу
Внепроцессный	Производительность ниже, чем во внутрипроцессном режиме. Надежность тоже не всегда соответствует требованиям, т. к. состояние сеансов сохраняется в памяти. Но данный режим более надежен по сравнению с внутрипроцессным, т. к. состоянием сеансов управляет отдельный процесс. Режим является масштабируемым

Таблица 16.2 (окончание)

Режим управления состоянием сеансов	Краткое описание
На основе SQL Server	Самая низкая производительность из всех рассматриваемых режимов, но обеспечивается наивысший уровень надежности. Режим является масштабируемым

Управление состояниями сеансов в ASP.NET

Управление сеансами в ASP.NET происходит с помощью файлов Web.config. При этом существует два уровня конфигурации:

- на уровне машины — связана с файлом Machine.config;
- на уровне приложения — связана с файлом Web.config.

Конфигурация на уровне приложения имеет более высокий приоритет по сравнению с конфигурацией на уровне машины.

Содержимое одного из файлов Web.config (ранее рассмотренного Web-сервиса `MyWebS1`, листинг 16.7) вы можете видеть в листинге 16.9.

Листинг 16.9. Содержимое файла Web.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

   <system.web>

   <compilation
      defaultLanguage="c#"
      debug="true"
   />
   <customErrors
   mode="RemoteOnly"
   />

   <authentication mode="Windows" />

   <trace
      enabled="false"
      requestLimit="10"
      pageOutput="false"
      traceMode="SortByTime"
      localOnly="true"
   />
```

```
    <sessionState
        mode="InProc"
        stateConnectionString="tcpip=127.0.0.1:42424"
        sqlConnectionString="data source=127.0.0.1;
        user id=sa;password="
        cookieless="false"
        timeout="20"
    />

    <globalization
        requestEncoding="utf-8"
        responseEncoding="utf-8"
    />

    </system.web>

</configuration>
```

Примечание

Текст, находящийся внутри файла Web.config является чувствительным к регистру.

Класс `SessionState`, который используется в листинге 16.9, отвечает непосредственно за управление состоянием сеанса. Основные свойства данного класса приведены в табл. 16.3.

Таблица 16.3. *Основные свойства класса* `SessionState`

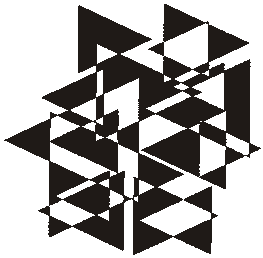
Свойство	Краткое описание
`mode`	Определяет режим отслеживания состояния сеанса. Может принимать следующие значения: `Off` — режим отслеживания отключен; `InProc` — внутрипроцессный режим (по умолчанию); `StateServer` — внепроцессный режим; `SQLServer` — сохранение данных сеанса в базе данных SQL Server
`stateConnectionString`	Указывает IP-адрес удаленного сервера (не SQL Server) и его порт. Используется только при внепроцессном режиме (`mode="StateServer"`)
`sqlConnectionString`	Позволяет задать строку соединения с SQL Server. Используется, только если `mode="SQLServer"`

Таблица 16.3 *(окончание)*

Свойство	Краткое описание
`cookieless`	Если данное свойство имеет значение `true`, то ASP.NET вставляет уникальный ключ в URL для навигации между страницами Web-приложения, вместо задания его в Cookie клиента (в противном случае использует Cookie клиента)
`timeout`	Указывает время ожидания для сеанса (в минутах). Отсчет начинается заново, после каждого нового запроса. По умолчанию установлено значение в 20 минут

Итак, в данной главе были рассмотрены возможности и преимущества технологии ASP.NET при создании Web-приложений. Эти возможности пригодятся вам при реализации своих собственных Web-приложений.

В *гл. 17* и *18* описывается создание Web-приложения, позволяющего осуществлять продажи компьютерной техники через Интернет, с использованием возможностей и технологий, предоставляемых средой разработки Microsoft Visual Studio .NET 2003.



ГЛАВА 17

Начало проекта

В этой главе рассматривается начало создания программы, которая позволит вам организовать собственный интернет-магазин по торговле компьютерной техникой.

Обзор ранних технологий

В начале развития торговли с помощью сети Интернет все начиналось с небольших статичных сайтов, единственной целью которых являлось привести потенциальных покупателей в существующий физический магазин. На таких сайтах первоначально клиенты могли найти лишь контактные телефоны, адрес, схему проезда и, в лучшем случае, электронный адрес.

Постепенно, с развитием систем управления базами данных и увеличением пропускных способностей каналов, начали появляться периодически обновляемые интернет-версии прайс-листов на товары или услуги компаний.

Как правило, под интернет-магазином тогда понимали именно интернет-интерпретацию реально существующего магазина, которая позволяла просматривать прайс-листы этого магазина посредством Интернета. Так появились онлайновые каталоги.

Далее следуют три этапа, обеспечивших полноценное существование интернет-магазинов, как полностью самостоятельных организаций. Перечислим эти этапы.

- Этап улучшения средств взаимодействия с клиентом. Клиент получил возможность не только просматривать, но и заказывать, а впоследствии и оплачивать товары или услуги, не выходя из-за компьютера.
- Этап разделения между реальными магазинами и интернет-магазинами, т. е. интернет-магазин начал существовать виртуально, без своего физического воплощения.

- Этап начала поставки товаров и услуг не только конечным потребителям, но и дистрибьюторам.

Постановка задачи

По сути дела, задача состоит в организации базы данных всего перечня товаров интернет-магазина, реализующего компьютеры и комплектующие. Далее, необходимо реализовать средства демонстрации товаров, позволить покупателю выбирать и добавлять товары в "корзину" и посылать заказ по электронной почте.

Но здесь есть некоторые тонкости. Далеко не все покупатели точно знают, что именно хотят приобрести, да и многие товары имеют различные специфические параметры. Таким образом, зачастую требуется консультация эксперта по товарам.

Попытаемся реализовать такого эксперта программно. Создадим несколько шаблонов типовых конфигураций персональных компьютеров, к примеру "Графическая станция", "Мультимедийный компьютер для дома", "Сервер баз данных" и др. Затем будем предлагать покупателям выбор наиболее подходящего шаблона, избавляя его тем самым от необходимости подбора комплектующих, в ходе которого им могут быть выбраны, например, два несовместимых устройства.

Для начала представим себе процесс покупки компьютера и попытаемся формализовать его. Приведенная ниже последовательность шагов описывает этот процесс следующим образом:

- для некомпетентных покупателей:
  - выбор типовой конфигурации;
  - просмотр заказа и его подтверждение;
  - регистрация заказа в системе, с посылкой его менеджеру виртуального магазина в отдел продаж или доставки;
- для компетентных покупателей:
  - выбор комплектующих и их количества;
  - просмотр заказа и его подтверждение;
  - регистрация заказа в системе, с посылкой его менеджеру виртуального магазина в отдел продаж или доставки.

Таким образом, последовательность выполняемых шагов по взаимодействию клиента с системой крайне проста, и в нашем случае, каждый из перечисленных шагов будет представлен отдельным ASP-файлом.

Создание и подготовка базы данных

Для создания базы данных (назовем ее `Shop`) нам понадобится программа Microsoft Access, входящая в состав Microsoft Office.

От процесса проектирования базы данных будет зависеть дальнейший объем программной логики нашего приложения.

При проектировании интернет-магазина будем стремиться к тому, чтобы страницы содержали как можно меньше строк и констант, выведем все, что можно, за их пределы, вплоть до наименований позиций, т. к. последние могут меняться довольно часто. Таким образом, мы добьемся от кода некоторой универсальности и независимости от данных.

Структура нашей базы данных показана на рис. 17.1.

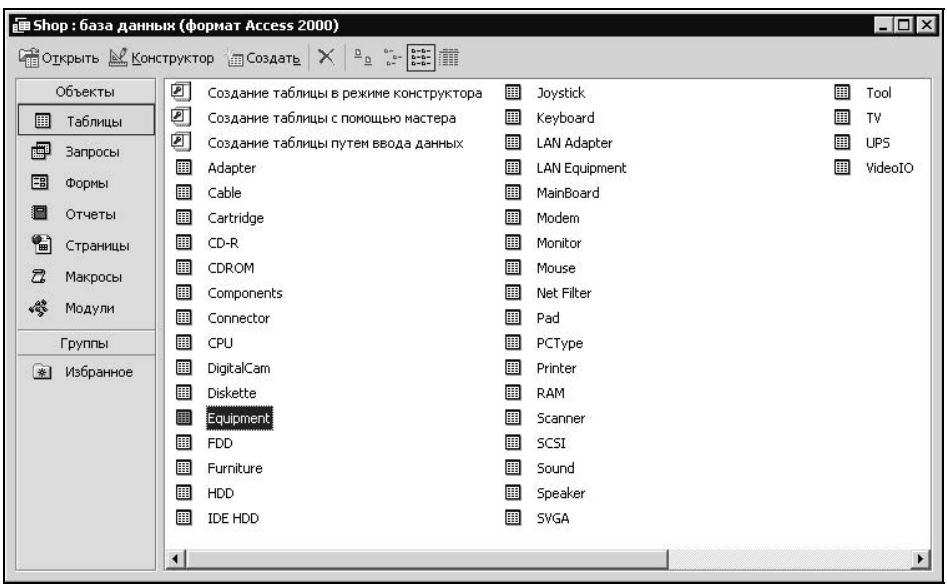


Рис. 17.1. Структура базы данных `Shop`

Создадим три управляющие таблицы:

- таблица `Components` — для хранения информации о наименованиях всех категорий и их соответствиях таблицам с позициями, ценами и условиями гарантий (рис. 17.2);
- таблица `Equipment` — с перечнем дополнительных категорий (рис. 17.3) для компетентных покупателей;
- таблица `PCType` — для некомпетентных покупателей, желающих приобрести компьютер с определенной целью (например, для дома или для офиса) (рис. 17.4).

Components : таблица

ID	CategoryName	CategoryTableName	IsPCComponent
1	Процессоры	CPU	1
2	Материнские платы	MainBoard	1
3	Видеоадаптеры	SVGA	1
4	Звуковые адаптеры	Sound	1
5	Винчестеры	HDD	1
6	Дисководы 3.5"	FDD	1
7	Принтеры	Printer	0
8	Мониторы	Monitor	1
9	Модули памяти	RAM	1
10	Приводы CD-ROM	CDROM	1
11	Платы видео вв./выв.	VideoIO	0
12	Колонки	Speaker	1
13	Клавиатуры	Keyboard	1
14	Мыши	Mouse	1
15	Модемы	Modem	1
16	Сетевые адаптеры	LAN Adapter	0
17	Сетевое оборудование	LAN Equipment	0

Запись: 15 из 32

Рис. 17.2. Таблица `Components`

Equipment : таблица

ID	Type
1	Сетевое оборудование
2	Компьютерная мебель
3	Расходные материалы
4	Кабели
5	Инструменты
6	Осветительные приборы
7	Оргтехника
(Счетчик)	

Запись: 1 из 7

Рис. 17.3. Таблица `Equipment`

PCType : таблица

ID	PCType	CPUID	MainboardID	SVGAID	HDDID	FDDID	CdromID
6	Компьютер для дома (обычный)	1	1	1	1	1	1
7	Компьютер для офиса (обычный)	1	1	1	1	1	1
8	Настольная издательская система	2	2	2	2	2	2
9	Студия видеомонтажа	2	2	2	2	2	2
10	Звуковая студия	1	1	1	1	1	1
11	Графическая станция	2	2	2	2	2	2
12	Сервер баз данных	3	1	1	1	1	1
13	Сервер приложений	2	2	2	2	2	2
15	Компьютер для дома (мультимедийный)	1	2	1	2	1	2
16	Компьютер для офиса (мультимедийный)	5	2	2	2	2	2
17	Компьютер для дома (мультимедийный + интернет)	1	1	1	1	1	1
(Счетчик)		0	0	0	0	0	0

Запись: 4 из 11

Рис. 17.4. Таблица `PCType`

Заметим, что поле `IsPCComponent` в таблице `Components`, имеющее булево значение, позволяет определить, является ли данный компонент обязательным составляющим персонального компьютера или нет. Этот факт понадобится нам в дальнейшем.

Все остальные таблицы создадим по одному и тому же шаблону с наименованием позиции (поле `Title`), розничной ценой (поле `Price1`), мелкооптовой ценой (поле `Price2`) и оптовой ценой (поле `Price3`), а также с условиями гарантии на товар (поле `Description`). Поля таблиц отображены на рис. 17.5.

MainBoard : таблица

ID	Title	Price1	Price2	Price3	Description
19	MB Asustek K7V-T VIA KX133 Slot-A (ATX,AGP4)	119,07	117,09	116,31	1 год
20	MB Asustek P2B-S 440BX Slot-1 UWSCSI(ATX,3D,AGP,4PCI,3ISA)	308,45	303,30	301,31	1 год
21	MB Asustek P3W-E i810E Slot-1 SVGA 4Mb UD66(ATX,3D,AMR,6PCI)	139,48	137,16	136,25	1 год
22	MB Atrend ATC 6230 440BX Slot-1(AT/ATX,3D,AGP,3PCI,2ISA)	117,94	115,97	115,21	6 мес
23	MB Chaintech 7ATA Slot-A SB (ATX,3D,AGP,5PCI,1ISA)	109,26	107,44	106,73	1 год
24	MB FORMOZA i810 F810 Socket-370 (ATX, SV, SOUND)	85,134	84,567	84,084	6 мес
25	MB Gigabyte 5AA ALI Socket-7(AT/ATX,3D,AGP,3PCI,2ISA)	73,71	72,482	72,009	1 год
26	MB Gigabyte 6VXE7+ VIA693A Socket-370 UD66(ATX,3D,AGP,5PCI,2ISA)	76,923	75,642	75,149	1 год
27	MB Gigabyte 6WMMC7 i810 Socket-370 SVGA+SB UD66(mATX,2D,AMR,3PCI)	83,349	81,963	81,417	1 год
28	MB Gigabyte 7IXE AMD751 Slot-A (ATX,3D,AGP,5PCI,2ISA)	99,918	98,249	97,608	1 год
29	MB Gigabyte DLX 440LX DualSlot-1 UWSCSI(ATX,AGP,4D,4PCI,3ISA)	165,38	164,27	163,33	1 год
30	MB SuperMicro P6DBU 440BX DualSlot-1 UW2SCSI(ATX,4D,AGP,5PCI,3ISA)	436,59	429,31	426,48	1 год
31	MB SuperMicro P6SBU 440BX Slot-1 Ultra2SCSI(ATX,4D,AGP,4PCI,3ISA)	294,84	289,93	288,02	1 год
32	MB SuperMicro SSM i815 Socket-370 (mATX,3D,AGP,2PCI,AMR)	149,69	147,19	146,22	1 год
33	MB SuperMicro SWD i810DC Slot-1 SVGA+SB UD66(mATX,2D, AMR,3PCI)	111,09	108,19	104,72	1 год
34	MB SuperMicro SWM i810 Socket-370 SVGA+SB(mATX,2D,1AMR,3PCI)	89,586	88,095	87,507	1 год
35	MB SuperPower 5BSM SIS530 Socket-7 SVGA+SB (AT/ATX,3D,3PCI,1ISA)	77,7	76,65	74,55	1 год
36	MB ZIDA BXV98-CT VIA693 Socket-370(AT/ATX,3D,AGP,4PCI,2ISA)	51,03	50,18	49,844	6 мес
37	MB ZIDA BXi98 440BX Slot-1(ATX,3D,AGP,5PCI,2ISA)	62,37	61,331	60,921	6 мес
38	MB ZIDA CX 3DCM SIS620 Socket-370 SVGA+SB(mATX,3D,3PCI,1ISA)	57,503	56,537	56,259	6 мес

Запись: 31 из 42

Рис. 17.5. Поля таблиц

Таблица `PCType` осуществляет взаимосвязь всех остальных таблиц и позволяет организовать системы предложения оптимальной конфигурации компьютера. Пример такой взаимосвязи представлен на рис. 17.6.

PCType : таблица

ID	PCType	CPUID	MainboardID	SVGAID	HDDID	FDDID	CdromID	SoundII
6	Компьютер для дома (обычный)	1	1	1	1	1	1	

Title	Price1	Price2	Price3	Description
1 CPU				

ID	PCType	CPUID	MainboardID	SVGAID	HDDID	FDDID	CdromID	SoundII
7	Компьютер для офиса (обычный)	1	1	1	1	1	1	

Title	Price1	Price2	Price3	Description
1 CPU				

ID	PCType	CPUID	MainboardID	SVGAID	HDDID	FDDID	CdromID	SoundII
8	Настольная издательская система	2	2	2	2	2	2	

Title	Price1	Price2	Price3	Description
CPU AMD ATHLON 550	77,763	76,461	75,957	1 год

Запись: 1 из 1

Рис. 17.6. Взаимосвязь таблиц

Каждая строка таблицы `PCType` содержит номера ключевых полей всех таблиц с позициями комплектующих.

Теперь пропишем нашу базу данных в соответствующем разделе источников данных системы. Для этого необходимо выполнить следующие действия:

1. Запустите программу-конфигуратор источников данных (**Data Sources ODBC**), **Пуск | Настройка | Панель управления | Администрирование | Источники данных ODBC**.
2. Перейдите на вкладку **Системный DSN** открывшегося диалогового окна и создайте новый источник данных, нажав на кнопке **Добавить**.
3. В появившемся списке драйверов выберите драйвер баз данных **Microsoft Access Driver** (*.mdb) и нажмите на кнопке **Готово**.
4. В строке **Имя источника данных** задайте имя базы данных, в нашем случае `Shop` (это имя, по которому мы в дальнейшем будем обращаться к ней).
5. Нажмите на кнопке **Выбрать**, выберите файл базы данных и нажмите кнопку **OK**.

Вы увидите появившуюся строку в списке источников данных системы (рис. 17.7).

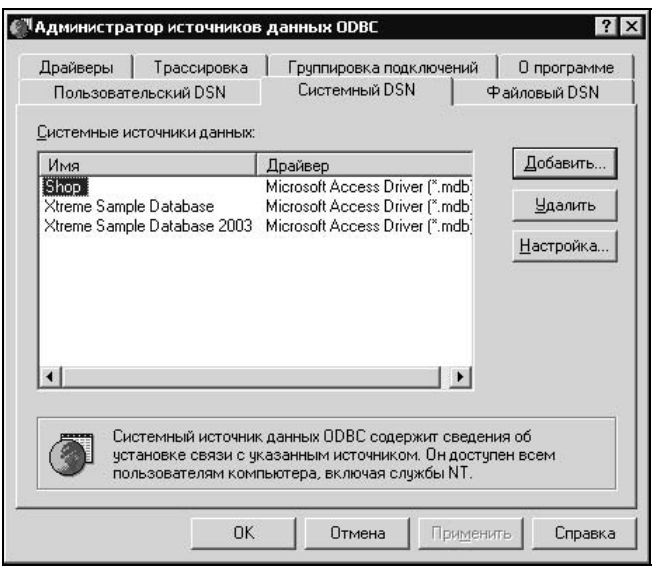
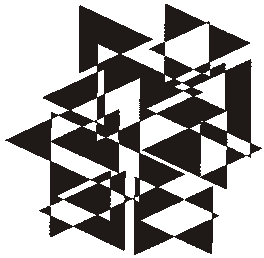


Рис. 17.7. Список источников данных

Теперь, когда база данных готова, можно приступать к созданию самого интернет-магазина. Этим мы займемся в *гл. 18*.

ГЛАВА 18



Создание проекта

В этой главе мы продолжим создавать собственный интернет-магазин. Рассмотрим структуру нашего интернет-магазина, а также код, который позволит ему работать.

Структура интернет-магазина

Рассмотрим структуру создаваемого интернет-магазина. Следует отметить, что и при автоматическом (по типовым шаблонам), и при самостоятельном подборе конфигурации загружается одна и та же форма MainForm.asp.

Однако если пользователь выбирал какую-нибудь типовую конфигурацию, то в последующей форме по умолчанию будут загружены позиции, соответствующие выбранной типовой конфигурации (из таблицы `PCtype`), в противном случае — первые позиции (в которых напишем "Выберите позицию").

При выборе раздела "Подбор дополнительного оборудования" загружается отдельная форма, но во всех остальных случаях ход работы программы одинаков:

- подтверждение заказа;
- сбор информации о покупателе;
- отправка заказа по электронной почте.

Итак, теперь можно приступать к написанию кода.

Главная страница — выбор типа заказа

Для начала создадим страницу, в которой пользователь сможет выбрать один из трех режимов дальнейшего взаимодействия с сайтом, а именно:

- покупка компьютера с помощью эксперта;
- самостоятельный выбор компьютера;
- приобретение дополнительного оборудования.

Код возможного примера такой страницы приведен в листинге 18.1.

Листинг 18.1. Страница index.asp

```
<html>
<head>
<title>Виртуальный магазин </title>
<meta http-equiv="Content-Type"
content="text/html; charset=windows-1251">
</head>

<body bgcolor="#006767" text="#FFFFFF" link="#FFFF00" vlink="#FFFF00">
<div align="center">
   <p>Добро пожаловать в наш виртуальный магазин </p>
   <p><a href="auto.asp">Подбор компьютера автоматически
   (с нашей помощью)</a></p>
   <p><a href="MainForm.asp?ID=2">Подбор компьютера
   самостоятельно</a></p>
   <p><a href="equip.asp">Покупка комплектующих</a></p>
   <p> </p>
</div>
</body>
</html>
```

Как вы можете видеть, пока все просто. Параметр `ID`, равный числу 2, передается в форму MainForm.asp, где он будет свидетельствовать о том, что пользователь решил выбирать состав компьютера самостоятельно.

Выбор типовой конфигурации компьютера

Теперь предложим пользователю выбрать типовую конфигурацию персонального компьютера, все данные берутся из таблицы `PCType`. Текст соответствующего запроса приведен в листинге 18.2.

Листинг 18.2. Файл auto.asp

```
<html>
<head>
<title>Виртуальный магазин </title>
```

```
<meta http-equiv="Content-Type"
content="text/html; charset=windows-1251">
</head>

<body bgcolor="#006767" text="#FFFFFF"
link="#CCFFCC" vlink="#FFFF00">
<div align="center">

<p>Подбор компьютера автоматически
(с нашей помощью)</p>
<p>Выберите шаблон конфигурации,
максимально соответствующий вашим требованиям:</p>

<form method="POST" action="MainForm.asp?ID=1">
<select name="PCType" size="1">

<%
   Set db = Server.CreateObject("ADODB.Connection")
   db.Open "DSN=Shop;UID=sa;PWD=;"

   SQLQuery = "Select * From PCType ORDER BY PCType"
   Set rs = db.Execute(SQLQuery)

   Do While NOT Rs.EOF
   Response.Write "<option value='" _
   & rs.Fields("ID").value & _
   "'>" _
   & rs.Fields("PCType").value & "</option>"
   Rs.MoveNext
   Loop

   ' в качестве элементов списка загрузим элементы поля PCType
   ' таблицы PCType базы данных Shop в цикле

   db.Close
   Set db = Nothing
%>

</select>

<input type="submit" name="Submit" value="&gt;&gt;">
</form>
</div>
</body>
</html>
```

Выбор дополнительного оборудования

Для выбора дополнительного оборудования используем текст листинга 18.3.

Листинг 18.3. Файл equip.asp

```
<html>
<head>
<title>Виртуальный магазин </title>
<meta http-equiv="Content-Type"
content="text/html; charset=windows-1251">
</head>

<body bgcolor="#006767" text="#FFFFFF" link="#CCFFCC" vlink="#FFFF00">
<div align="center">

<p>Покупка комплектующих</p>
<p>Выберите тип комплектующих</p>

<form method="get" action="EquipForm.asp">

<select name="select" size="1">

<%
   Set db = Server.CreateObject("ADODB.Connection")
   db.Open "DSN=Shop;UID=sa;PWD=;"

   SQLQuery = "Select * From Equipment ORDER BY Type"
   Set rs = db.Execute(SQLQuery)

   Do While NOT Rs.EOF
   Response.Write "<option value='" _
   & rs.Fields("ID").value & _
   "'>" _
   & rs.Fields("Type").value & "</option>"
   Rs.MoveNext
   Loop

   ' в качестве элементов списка загрузим элементы поля Type
   ' таблицы Equipment базы данных Shop в цикле

   db.Close
   Set db = Nothing
%>

<input type="submit" name="Submit" value="&gt;&gt;">
</form>
```

```
</div>
</body>
</html>
```

Подбор комплектующих

Нам предстоит организовать средство просмотра и выбора позиций, соответствующих именам таблиц из определенного поля таблицы `Components`, где параметр `IsPCComponent` равен 1. Необходимо осуществить возможности указания количества единиц комплектующих каждой категории и просмотра цены за единицу, и обеспечить вычисление суммарной цены (за весь набор), причем в интерактивном режиме. Это означает, что, выбрав из списка ту или иную позицию, пользователь должен сразу же увидеть справа цену за ее единицу и обновленную цену за весь набор, с учетом последнего изменения. То же самое должно происходить при изменении количества единиц каждой позиции.

Для технической реализации этого визуально в пределах одной формы придется воспользоваться HTML-тегом `iframe`, причем скрытым, а в нужных местах, в теле формы, использовать тег `b`, например, `<b id="Message1"></b>`, где в переменную `Message1` с помощью JavaScript-функций будет загружаться требуемое значение.

Для этого необходимо загрузить значения цены, причем для каждой категории комплектующих, например, таким образом:

```
...
function OnLoadMess1()
{
   Message1.innerText = FrmPrice.document.all.Message1.innerText;
   //  загрузка в переменную Message1 текста из фрейма FrmPrice
}
...
function GetPrice1 ()
{
   var content ="ifrsrc/ifrsrc1.asp?mess=" +
   document.forms.MainForm.elements.select1.value;
   //  формирование строки — ссылки на скрипт,
   //  извлекающий значение цены данной позиции,
   //  заданной категории комплектующих
   FrmPrice1.document.URL=content;
   //  передача управления (переход к) скрипту
   OnLodTotalPrice();
   //  пересчет суммарной стоимости всего набора
   }
...
```

Сам же скрипт, извлекающий значение цены, может выглядеть, например, следующим образом:

```
<%@ Language=VBScript %>
<html>
<head>
<SCRIPT Language=JavaScript>
function OnLoad()
{
   window.parent.OnLoadMess1();
}
</SCRIPT>
</head>

<body onload="OnLoad()">
<b id=Message1>
<%
   Set dbo = Server.CreateObject("ADODB.Connection")
   dbo.Open "DSN=Shop;UID=sa;PWD=;"
   ' получим значение, переданное в скрипт формой "MainForm.asp"
   Title = Request.QueryString("mess")

   ' выберем ту позицию из необходимой таблицы,
   ' где значение наименования позиции совпадает с переданным значением

   SQLQuery = "Select * From SCSI WHERE Title = '" & Title & "'"
   Set rso = dbo.Execute(SQLQuery)
   Response.Write rso.Fields("Price1").value
   dbo.Close
   Set dbo = Nothing
%>

</b>
</body>
</html>
```

Как вы можете видеть, скрипт содержит в себе единственную JavaScript-функцию, вызываемую при его загрузке. Данная функция вызывает соответствующую функцию скрипта родителя (т. е. MainForm.asp). JavaScript-функция срабатывает на клиенте, а не на сервере и, следовательно, в этот момент значение цены выбранной позиции (переданной скрипту в переменной `mess`) уже извлечено из соответствующей таблицы (в данном случае таблицы `SCSI`) и выведено на экран.

Очевидно, что эти действия должны быть проделаны над всем множеством категорий и для каждой категории должен быть написан свой скрипт — загрузчик цены за единицу выбранной позиции.

Теперь настало время подготовить главную форму, в которой и будет осуществляться основная часть взаимодействия магазина с покупателем. Как было сказано выше, в эту форму передается вся предыстория, по которой потенциальный покупатель добрался до нее. Напомню, что делается это посредством переменной `ID`, передаваемой форме в качестве параметра (листинг 18.4).

Листинг 18.4. Файл MainForm.asp

```
<html>
<head>
<title>Виртуальный магазин </title>
<meta http-equiv="Content-Type"
content="text/html; charset=windows-1251">

<SCRIPT>

<%
   Set db = Server.CreateObject("ADODB.Connection")
   db.Open "DSN=Shop;UID=sa;PWD=;"

   SQLQuery = "Select * From Components WHERE IsPCComponent = 1" & _
   " ORDER BY CategoryName ASC"

   Set rs = db.Execute(SQLQuery)

   CurrentCategory = 0

   Do While NOT rs.EOF
   CurrentCategory = CurrentCategory + 1

   Response.Write "function OnLoadMess" _
   & CurrentCategory & "(){"
   Response.Write "Message" & CurrentCategory _
   & ".innerText=FrmPrice" & CurrentCategory & _
   ".document.all.Message" _
   & CurrentCategory & ".innerText;}"
   Response.Write "function GetPrice" _
   & CurrentCategory & "(){"
   Response.Write "var content = 'ifrsrc/ifrsrc" _
   & CurrentCategory &".asp?mess='" & _
   "+document.forms.MainForm.elements.select" _
   & CurrentCategory & ".value;"

   Response.Write "FrmPrice" _
   & CurrentCategory & ".document.URL=content;"
```

```
    Response.Write "OnLodTotalPrice();}"
   rs.MoveNext
   Loop

   db.Close
   Set db = Nothing
%>

function OnLodTotalPrice()    //  вычисление суммарной цены
{
   var Total;

   Total = 0;

   //  вычисление суммарной цены производится суммированием
   //  всех цен, умноженных на количество выбранных единиц
   //  каждой позиции соответственно

   Total = Total + parseInt(Message1.innerText)
   * (document.forms.MainForm.elements.quant1.value);
   Total = Total + parseInt(Message2.innerText)
   * (document.forms.MainForm.elements.quant2.value);
   Total = Total + parseInt(Message3.innerText)
   * (document.forms.MainForm.elements.quant3.value);
   Total = Total + parseInt(Message4.innerText)
   * (document.forms.MainForm.elements.quant4.value);
   Total = Total + parseInt(Message5.innerText)
   * (document.forms.MainForm.elements.quant5.value);
   Total = Total + parseInt(Message6.innerText)
   * (document.forms.MainForm.elements.quant6.value);
   Total = Total + parseInt(Message7.innerText)
   * (document.forms.MainForm.elements.quant7.value);
   Total = Total + parseInt(Message8.innerText)
   * (document.forms.MainForm.elements.quant8.value);
   Total = Total + parseInt(Message9.innerText)
   * (document.forms.MainForm.elements.quant9.value);
   Total = Total + parseInt(Message10.innerText)
   * (document.forms.MainForm.elements.quant10.value);
   Total = Total + parseInt(Message11.innerText)
   * (document.forms.MainForm.elements.quant11.value);
   Total = Total + parseInt(Message12.innerText)
   * (document.forms.MainForm.elements.quant12.value);
   Total = Total + parseInt(Message13.innerText)
   * (document.forms.MainForm.elements.quant13.value);
```

```
   Total = Total + parseInt(Message14.innerText)
   * (document.forms.MainForm.elements.quant14.value);
   Total = Total + parseInt(Message15.innerText)
   * (document.forms.MainForm.elements.quant15.value);

   TMessage.innerText = String(Total);
}

  //  установка значений по умолчанию при загрузке страницы
function Set()
{
   GetPrice1();
   GetPrice2();
   GetPrice3();
   GetPrice4();
   GetPrice5();
   GetPrice6();
   GetPrice7();
   GetPrice8();
   GetPrice9();
   GetPrice10();
   GetPrice11();
   GetPrice12();
   GetPrice13();
   GetPrice14();
   GetPrice15();

   SetQuantity();
   OnLodTotalPrice();
}

  //  реакция на изменение селекторов количества
function SetQuantity()
{
   TotalPrice.document.URL = "tprice.asp?mess=aaa";
}

</SCRIPT>
</head>

<body bgcolor="#006767" text="#FFFFFF"
   link="#FFFF00" vlink="#FFFF00"
   onLoad="Set();">
<div align="center">
```

```
<%

    'Соединяемся с базой данных "Shop"

    Set db = Server.CreateObject("ADODB.Connection")
    db.Open "DSN=Shop;UID=sa;PWD=;"

    TheID = Request.QueryString("ID")

    If TheID = 1 Then
    Response.Write "<p>Подбор компьютера " _
    & "автоматически (с нашей помощью)</p>"
    PCType = Request.Form("PCType")
    ASQLQery = "Select * From PCType WHERE ID = " & PCType
    Set result = db.Execute(ASQLQery)

    ' инициализация переменных, которые будут загружены
    ' в списки выбора по умолчанию (извлекаются
    ' из предыдущей формы)
    ' в случае автоматического подбора конфигурации

    CPUID = result.Fields("CPUID")
    MainBoardId = result.Fields("MainBoardId")
    SVGAID= result.Fields("SVGAID")
    HDDID = result.Fields("HDDID")
    FDDID = result.Fields("FDDID")
    CdromID = result.Fields("CdromID")
    SoundID = result.Fields("SoundID")
    ModemID = result.Fields("ModemID")
    KeybordID = result.Fields("KeyboardID")
    MouseID = result.Fields("MouseID")
    MonitorID = result.Fields("MonitorID")
    PadID = result.Fields("PadID")
    SCSIID= result.Fields("SCSIID")
    SpeakerID = result.Fields("SpeakerID")

    D = Date()
    D = FormatDateTime(D,2)

    str = "<p>Рекомендуемая конфигурация компьютера типа <br>'"
    str = str & result.Fields("PCType").value & "'"
    str = str & " по состоянию на: " & D & "</p>"
    Response.Write str

    result.Close
    Set result = Nothing
```

```
   Else

   Response.Write "<p>Подбор компьютера самостоятельно</p>"

   ' инициализация переменных, которые будут загружены
   ' в списки выбора по умолчанию (извлекаются из предыдущей формы)
   ' в случае самостоятельного подбора конфигурации

   CPUID = 1
   MainBoardId = 1
   SVGAID= 1
   HDDID = 1
   FDDID = 1
   CdromID = 1
   SoundID = 1
   ModemID = 1
   KeybordID = 1
   MouseID = 1
   MonitorID = 1
   PadID = 1
   SCSIID= 1
   SpeakerID = 1

   End If
%>

<form method="get" name= "MainForm" action="MainFormProc.asp">

<table width="457" border="2" cellspacing="0" cellpadding="0">
<tr>
<td width="223"><b>Наименование позиции</b></td>
<td width="80"><b>Позиция</b></td>
<td width="97"><b>Количество</b></td>
<td width="63"><b>Цена</b></td>
</tr>

<%

   ' Формируем SQL-запрос — все категории, являющиеся обязательными
   ' компонентами персонального компьютера, упорядоченные по алфавиту

   SQLQuery = "Select * From Components WHERE
   IsPCComponent = 1 ORDER BY CategoryName ASC"
   Set rs = db.Execute(SQLQuery)

   CurrentCategory = 0
   TotalPrice = 0
```

```
' в цикле по всем категориям

Do While NOT rs.EOF
CurrentCategory = CurrentCategory + 1 ' счетчик категорий

Response.Write "<tr><td>"
Response.Write "" _
& rs.Fields("CategoryName").value & ": "
Response.Write "</td><td>"

' создаем селектор "select#"
' определяем функцию-реакцию на его изменение "GetPrice#"
' где # — значение счетчика категорий для каждой категории

Response.Write "<select id=" & "'select" _
& CurrentCategory & "' name='select" &CurrentCategory & _
"' onchange='GetPrice" & CurrentCategory & "()'"

Response.Write " style='WIDTH: 400px'>"

CurTableName = rs.Fields("CategoryTableName").value

' зафиксируем имя текущей таблицы — переменной "CurTableName"

' определим, какая позиция должна быть сделана активной (по умолчанию)

If CurTableName = "CPU" Then
CurTemplateSelectID = CPUID
End If
If CurTableName = "Mainboard" Then
CurTemplateSelectID = MainboardID
End If
If CurTableName = "SVGA" Then
CurTemplateSelectID = SVGAID
End If
If CurTableName = "HDD" Then
CurTemplateSelectID = HDDID
End If
If CurTableName = "FDD" Then
CurTemplateSelectID = FDDID
End If
If CurTableName = "Cdrom" Then
CurTemplateSelectID = CdromID
End If
If CurTableName = "Sound" Then
CurTemplateSelectID = SoundID
End If
```

```
If CurTableName = "Modem" Then
CurTemplateSelectID = ModemID
End If
If CurTableName = "Keyboard" Then
CurTemplateSelectID = KeyboardID
End If
If CurTableName = "Mouse" Then
CurTemplateSelectID = MouseID
End If
If CurTableName = "Monitor" Then
CurTemplateSelectID = MonitorID
End If
If CurTableName = "Pad" Then
CurTemplateSelectID = PadID
End If
If CurTableName = "SCSI" Then
CurTemplateSelectID = SCSIID
End If
If CurTableName = "Speaker" Then
CurTemplateSelectID = SpeakerID
End If

' создадим список позиций для каждой категории,
' упорядоченный по алфавиту
' извлечение будем производить из текущей таблицы —
' переменной "CurTableName"

ISQLQuery = "Select * From " & CStr (CurTableName)
& " ORDER BY Title ASC"
Response.Write ISQLQuery
Set res = db.Execute(ISQLQuery)

CurrentItem = 0

Do While NOT res.EOF
CurrentItem = CurrentItem + 1

' заполним в цикле каждый селектор значениями позиций
' из соответствующей таблицы
str = "<option"

' если текущая позиция должна быть позицией по умолчанию — сделаем это

If CurrentItem = CurTemplateSelectID Then
str = str & " selected"
TemplatePriceValue = res.Fields("Price1").value
End If
```

```
str = str & " value='" _
& res.Fields("Title").value & _
"'>" & res.Fields("Title").value _
& "</option>"

Response.Write str
res.MoveNext
Loop

Response.Write "</select>"
res.Close
Set res = Nothing

Response.Write "</td><td>"

' реализуем селектор количества и заполним его числами от 1 до 50
' при изменении будет вызываться функция "SetQuantity()"

Response.Write "<select id='quant" _
& CurrentCategory & "' name='quant" & CurrentCategory &_
"' style='WIDTH: 50px' OnChange = 'SetQuantity()'>"

I = 1
Do While I <> 50
Response.Write "<option value = " & I _
& ">" & I & "</option>"
I = I + 1
Loop
Response.Write"</select>"

' определим невидимый фрейм типа "Iframe"
' с именем "FrmPrice#" для вычисления значений цен,
' где # — значение счетчика категорий
' для каждой категории

Response.Write "</td><td><IFRAME id='FrmPrice" _
& CurrentCategory & _
"' style='WIDTH: 80px' style='HEIGHT: 40px' style=
'display: none' name='FrmPrice" _
& CurrentCategory & _
"'></IFRAME>"

Response.Write "<b id='Message" & CurrentCategory _
&"'></b></td></tr>"
```

```
   ' определим скрытое поле для передачи значений цен
   ' в последующие скрипты

   Response.Write "<input type='hidden' name='price" _
   & CurrentCategory & "' value='" _
   & TemplatePriceValue &
   "'>"

   Rs.MoveNex

   ' суммируем цены для вычисления цены за весь набор
   TotalPrice = TotalPrice + TemplatePriceValue

   Loop

   db.Close
   Set db = Nothing

   Response.Write "<tr><td><b><i>Итого:
   </b></i></td><td> — </td><td> — </td>"
 Response.Write "<td><b><i>"

   ' определим невидимый фрейм типа "Iframe"
   ' с именем "TotalPrice" для вычисления суммы цен

   Response.Write "<IFRAME id='TotalPrice' " _
   & " style='WIDTH: 80px' style='HEIGHT: 40px" & _
   "' style='display: none' name='TotalPrice'></IFRAME>"

   Response.Write "<b id='TMessage'></b></b></i></td></tr>"
%>

</table>

<p>
<input type="submit" name="Submit" value="Продолжить">
<input type="reset" name="Reset" value="Очистить форму">
</p>

</form>
</div>
</body>
</html>
```

Подтверждение заказа (файл MainformProc.asp)

```
<html>
   <head>
   <meta http-equiv="Content-Type" content="text/html;
```

```
charset=windows-1251">
</head>

<body bgcolor="#006767" text="#FFFFFF"
link="#FFFF00"  vlink="#FFFF00">
<div align="center">

<form method="get" action="EmailOrder.asp">

<p>Вы заказываете:</p>

<table width="457" border="2" cellspacing="0"  cellpadding="0">

<tr>
<td width="25"><b>#</b></td>
<td width="350"><b>Позиция</b></td>
<td width="25"><b>Количество</b></td>
<td width="40"><b>Цена</b></td>
</tr>

<%
   I = 1

   Do While I <= 15

   Response.Write "<tr><td>" & CStr(I)

   ' прочитаем в цикле значения селекторов "select#"
   ' и сгенерируем скрытые элементы формы
   ' для передачи выбранных значений в последующие скрипты

   Str = "select"
   Str = Str & CStr(I)
   Res = Request.QueryString(Str)
   Response.Write "</td><td>" & Res
   Response.Write "<input type='hidden' name='" _
   & Str & "' value='" & Res & "'>"

   ' прочитаем в цикле значения селекторов "quant#"
   ' и сгенерируем скрытые элементы формы
   ' для передачи выбранных значений в последующие скрипты

   Str = "quant"
   Str = Str & CStr(I)
   Res = Request.QueryString(Str)
   Response.Write "</td><td>" & Res
   Response.Write "<input type='hidden' name='" _
   & Str & "' value='" & Res & "'>"
```

```
      ' прочитаем в цикле значения скрытых полей "pricet#"
      ' и сгенерируем скрытые элементы формы
      ' для передачи выбранных значений в последующие скрипты

      Str = "price"
      Str = Str & CStr(I)
      Res = Request.QueryString(Str)
      Response.Write "</td><td>" & Res
      Response.Write "<input type='hidden' name='" _
      & Str & "' value='" & Res & "'>"

      Response.Write "</td></tr>"

      I = I + 1
      Loop
   %>

   </table>
   <br>
   <input type="submit" name="Submit" value="Заказать">
   <input type="reset" name="Reset" value="Очистить  форму">

   </form>
   </div>
   </body>
</html>
```

Сбор данных о покупателе (файл EmailOrder.asp)

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
</head>

<body bgcolor="#006767" text="#FFFFFF"
link="#FFFF00" vlink="#FFFF00">
<div align="center">

<p>Отправка заказа по электронной почте:</p>

<p>Как с вами связаться</p>

<Form Method="get" Action = "Done.asp">
<table>
<tr>
<td>Ваше имя:</td>
<td><Input Type="text" Name="Name" Size="30"></Input></td>
</tr>
```

```
<tr>
<td>Ваш телефон:</td>
<td><Input Type="text" Name="Phone" Size="30"></Input></td>
</tr>

<tr>
<td>Ваш адрес:</td>
<td><Input Type="text" Name="Address" Size="30"></Input></td>
</tr>

<tr>
<td>Ваш электронный адрес:</td>
<td><Input Type="text" Name="Email" Size="30"></Input></td>
</tr>

<tr>
<td>Ваш UIN:</td>
<td><Input Type="text" Name="UIN" Size="30"></Input></td>
</tr>

<tr>
<td align = center><Input Type="submit"
Value="Отправить заказ"></Input></td>
</tr>

</table>

<%
   I = 1

   Do While I <= 15

   ' прочитаем в цикле значения скрытых полей "select#"
   ' и сгенерируем скрытые элементы формы
   ' для передачи выбранных значений в последующие скрипты

   Str = "select"
   Str = Str & CStr(I)
   Res = Request.QueryString(Str)
   Response.Write "<input type='hidden' name='" _
    & Str & "' value='" & Res & "'>"

   ' прочитаем в цикле значения скрытых полей "quant#"
   ' и сгенерируем скрытые элементы формы
   ' для передачи выбранных значений в последующие скрипты

   Str = " quant"
   Str = Str & CStr(I)
```

```
   Res = Request.QueryString(Str)
   Response.Write "<input type='hidden' name='" _
   & Str & "' value='" & Res & "'>"

   ' прочитаем в цикле значения скрытых полей "price#"
   ' и сгенерируем скрытые элементы формы
   ' для передачи выбранных значений в последующие скрипты

   Str = "price"
   Str = Str & CStr(I)
   Res = Request.QueryString(Str)
   Response.Write "<input type='hidden' name='" _
   & Str & "' value='" & Res & "'>"

   I = I + 1
   Loop
%>

</Form>
<br>
</div>
</body>
</html>
```

Отправка заказа по e-mail (файл Done.asp)

После того как все действия выполнены успешно, необходимо сформировать электронное письмо, которое должно быть послано в подтверждение выполненного заказа менеджеру по продажам или службе доставки нашего виртуального магазина, и самому пользователю одновременно. Для этого воспользуемся бесплатно поставляемым ActiveX-компонентом ASP E-mail 4.4, компании Persist Software (листинг 18.5).

Листинг 18.5. Файл Done.asp

```
<html>
<head>
<title>Виртуальный магазин </title>
<meta http-equiv="Content-Type"
content="text/html; charset=windows-1251">
</head>

<body bgcolor="#006767" text="#FFFFFF"
link="#FFFF00" vlink="#FFFF00">
```

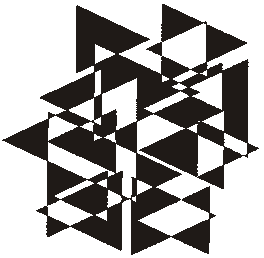
```
<%
   ' запросим значения, введенные пользователем

   Name= Request.QueryString("Name")
   Phone = Request.QueryString("Phone")
   Address = Request.QueryString("Address")
   Email = Request.QueryString("Email")
   UIN = Request.QueryString("UIN")

   ' создадим экземпляр объекта "Persits.MailSender"

   Set Mail = Server.CreateObject("Persits.MailSender")
   Mail.Host = "mail.compress.ru" 'Адрес SMTP сервера
   Mail.From = ponamarev@mail.ru' Адрес "от кого"
   Mail.FromName = "Vyacheslav Ponamarev"'Имя"от кого"

   Mail.AddAddress "ponamarev@mail.ru",
   "Vyacheslav Ponamarev" ' Адрес и имя кому
   Mail.AddAddress Email, Name ' Адрес и имя кому

   Mail.IsHTML = True' Посылка почты в формате HTML
   Mail.Subject = "PC Order!"' Поле Subject

   ' Формируем HTML-документ — тело нашего письма

   Body = "<HTML><HEAD>"
   Body = Body & "<meta http-equiv='Content-Type' content='text/html; "
   Body = Body & "charset=windows-1251'></HEAD><BODY>"
   Body = Body & "Уважаемый " & Name & "!<br><br>"
   Body = Body & "********************<br>"
   Body = Body & "Имя: " & Name & "<br>"
   Body = Body & "Телефон: " & Phone & "<br>"
   Body = Body & "Адрес: " & Address & "<br>"
   Body = Body & "E-mail: " & Email & "<br>"
   Body = Body & "UIN: " & UIN & "<br>"
   Body = Body & "********************" & "<br>"
   Body = Body & "Вы заказали: " & "<br>"

   I = 1

   Do While I <= 15

   Body = Body & I & ". "
   Str = "select"
   Str = Str & CStr(I)
   Res = Request.QueryString(Str)
   Body = Body & Res & " — "
```

```
   Str = "quant"
   Str = Str & CStr(I)
   Res = Request.QueryString(Str)
   Body = Body & Res & " — "

   Str = "price"
   Str = Str & CStr(I)
   Res = Request.QueryString(Str)
   Body = Body & Res & "<br>"
   I = I + 1
   Loop

   Body = Body & "********************" & "<br>"
   Body = Body & "Наши менеджеры свяжутся с вами в ближайшее "
   Body = Body & " время, по оставленным вами координатам."
   Body = Body & "<br>С наилучшими пожеланиями<br>"
   Body = Body & "<br>Сервер 'Виртуального магазина'"
   Body = Body & "</BODY></HTML>"

   Mail.Body = Body
   'Response.Write Mail.Body

   ' попытаемся послать наше письмо

   On Error Resume Next
   Mail.Send
   If Err <> 0 Then
   Response.Write "Ошибка передачи: " & Err.Description
   End If

   Response.Write "<center>Уважаемый " & Name & "!<br>"
   Response.Write "Ваш заказ был успешно отправлен "
   Response.Write "на сервер виртуального магазина.<br>"
   Response.Write "Спасибо!</center>"
%>

</body>
</html>
```

Итак, мы рассмотрели механизм создания интернет-магазина средствами ASP, которые нашли отличную поддержку в Microsoft Visual Studio .NET 2003.

В заключительной *гл. 19* рассказывается о принципах отладки готовых приложений.

ГЛАВА 19



Принципы отладки приложений

В этой главе речь пойдет об отладке созданных вами приложений с помощью средств, предоставляемых средой Visual Studio .NET 2003. Рассматриваются различные методы отладки и классы, которые используются при отладке приложений во время выполнения.

Простейшая отладка

Пространство имен `System.Diagnostics` содержит два основных класса, которые используются при отладке во время выполнения приложения:

- класс `Debug` — предназначен для отладки внутри среды разработки. Для активирования класса нужна директива `#define DEBUG` или соответствующая опция командной строки (`/d:DEBUG`);
- класс `Trace` — не требует директив или опций командной строки. Он активируется автоматически и применяется для программ, которые обладают функциями отладки. Использование данного класса увеличивает расход системных ресурсов.

Итак, если ваша программа не будет выдавать отладочную информацию после компиляции, то следует использовать класс `Debug`. В противном случае (если позволяют системные ресурсы), лучше использовать класс `Trace`.

Самый простейший способ проведения отладки во время выполнения приложения — это вывод необходимой информации на экран. Для этого вы можете использовать класс `Debug`, который содержит два метода вывода отладочной информации на экран: `Write()` и `WriteLine()`.

В листинге 19.1 приведен пример использования метода `WriteLine()` класса `Debug` (на языке C#).

Листинг 19.1. Пример с использованием класса `Debug`

```
#define DEBUG

using System;
using System.Diagnostics;

class MyDebug
{
   static void MyMethod()
   {
      //  вызов метода WriteLine() класса Debug
      Debug.WriteLine("Отладка");
   }
   static void Main(string[] args)
   {
      //  создание класса TextWriterTraceListener,
      //  который будет направлять
      //  вывод отладочной информации на экран
      TextWriterTraceListener myListener =
      new TextWriterTraceListener(Console.Out);
      //  добавление объекта myListener в коллекцию Debug.Listeners
      Debug.Listeners.Add(myListener);
      //  вызов метода MyMethod(), который будет объектом отладки
      MyMethod();
   }
}
```

Если выполнить данный пример, то на экран будет выведено слово:

```
Отладка
```

В примере из листинга 19.1 используется директива `#define DEBUG` для включения механизма отладки. Вы можете воспользоваться и другим методом — с помощью опции командной строки:

```
csc /d:DEBUG имя_файла.cs
```

Отмечу, что способность программы включать и выключать режим отладки носит название *условной отладки* (conditional debugging).

Пространство имен `System.Diagnostics` содержит атрибуты и ключи для включения или отключения режима отладки. В листинге 19.2 вы можете видеть пример использования атрибута `Conditional`.

Листинг 19.2. Пример использования атрибута `Conditional`

```
#define DEBUG

using System;
using System.Diagnostics;

class MyDebug
{
   static bool Debugging = true;
   [Conditional("DEBUG")]
   static void SetDebugListener()
   {
      TextWriterTraceListener myListener =
      new TextWriterTraceListener(Console.Out);
      Debug.Listeners.Add(myListener);
   }
   [Conditional("DEBUG")]
   static void CheckSt()
   {
      Debug.WriteLineIf(Debugging, "Отладка");
   }
   static void Main(string[] args)
   {
      SetDebugListener();
      CheckSt();
   }
}
```

Атрибут `Conditional` в данном примере помещен в первую строку объявления метода, который по желанию можно включать и отключать.

Рассмотрим первый параметр метода `WriteLineIf()`. Если он равен `true`, то вывод отладочной информации будет осуществляться, в противном случае — не будет.

В рассмотренных примерах (листинг 19.1 и 19.2) предполагается, что для того чтобы включить или выключить режим отладки, необходимо полностью перекомпилировать код приложения. Это не представляет особого труда во время создания программы. Но в процессе эксплуатации перекомпилировать программу всякий раз очень трудоемко. Чтобы этого избежать, вы можете использовать классы `Trace` и `BooleanSwitch`. Пример использования этих классов приведен в листинге 19.3.

Листинг 19.3. Пример использования классов `Trace` и `BooleanSwitch`

```
using System;
using System.Diagnostics;

class MySwitch
{
   BooleanSwitch traceOutput = new BooleanSwitch("TraceOutput", "Demo");
   void SetDebugListener()
   {
      TextWriterTraceListener myListener =
      new TextWriterTraceListener(Console.Out);
      Trace.Listeners.Add(myListener);
   }
   void CheckSt()
   {
      Trace.WriteLineIf(traceOutput.Enabled, "Отладка");
   }
   static void Main(string[] args)
   {
      MySwitch ms = new MySwitch();
      ms.SetDebugListener();
      ms.CheckSt();
   }
}
```

Как вы заметили, метод `WriteLineIf()` в данном примере использует свойство `Enabled` объекта `BooleanSwitch` в качестве первого параметра.

Для активирования режима трассировки нужно в файл конфигурации программы добавить соответствующую запись. Имя файла конфигурации должно совпадать с именем исполняемого файла и иметь расширение config.

Трассировка времени исполнения

Трассировкой времени исполнения (runtime tracing) называется возможность слежения за ходом выполнения программы.

Рассмотрим класс `TraceSwitch`, который похож на класс `BooleanSwitch`. Класс `TraceSwitch` позволяет задавать более высокую степень детализации для отображаемой отладочной информации.

В листинге 19.4 приведен пример использования класса `TraceSwitch`.

Листинг 19.4. Пример использования класса `TraceSwitch`

```
using System;
using System.Diagnostics;

class MyTraceSwitch
{
   public static TraceSwitch traceOutput =
   new TraceSwitch("TraceOutput", "Demo");
   void SetDebugListener()
   {
      TextWriterTraceListener myListener =
      new TextWriterTraceListener(Console.Out);
      Trace.Listeners.Add(myListener);
   }
   void CheckSt()
   {
      Trace.WriteLineIf(traceOutput.TraceInfo, "Отладка");
   }
   static void Main(string[] args)
   {
      MyTraceSwitch mts = new MyTraceSwitch();
      mts.SetDebugListener();
      mts.CheckSt();
   }
}
```

Вы, наверное, заметили, что первый параметр метода `WriteLineIf()` в данном примере представляет собой свойство `TraceInfo` класса `TraceSwitch`. Этот параметр может принимать одно из пяти возможных значений, которые соответствуют членам перечисления `TraceLevel`:

- `Error`, `(1)` — вывод сообщений об ошибках;
- `Info`, `(3)` — вывод информации, предупреждений и сообщений об ошибках;
- `Off`, `(0)` — отключение вывода информации;
- `Verbose`, `(4)` — вывод полной информации (все, что возможно);
- `Warning`, `(2)` — вывод предупреждений и сообщений об ошибках.

Значение данного параметра для класса `TraceSwitch` должно быть установлено в файле конфигурации. Диапазон допустимых значений от 0 до 4 (они указаны в списке в скобках).

Создание проверок

Часто в процессе отладки необходимо проверять состояние программы на предмет логической непротиворечивости через различные временные интервалы. Для этого вы можете использовать метод `Assert()` класса `Debug`. Допустимо расставить методы `Assert()` в критических местах программы и проверить ее (программы) логическую непротиворечивость.

Если формальное утверждение возвращает значение `false`, то оно будет отображено в окне сообщений.

В листинге 19.5 приведена простая программа, которая показывает работу с методом `Assert()`.

Листинг 19.5. Пример работы метода `Assert()`

```
using System;
using System.Diagnostics;

class MyAssert
{
   public static void Main(string[] args)
   {
      decimal a = -0.002m;
      Debug.Assert(a>=0.0m, "Ошибка в логике программы!");
   }
}
```

В данной программе приведен фрагмент, который никогда не должен возвращать отрицательный результат.

Метод `Assert()` принимает два параметра. Первый параметр — условие, которое возвращает логическое значение. Второй параметр — отображаемое сообщение, в случае, если условие принимает значение `false`.

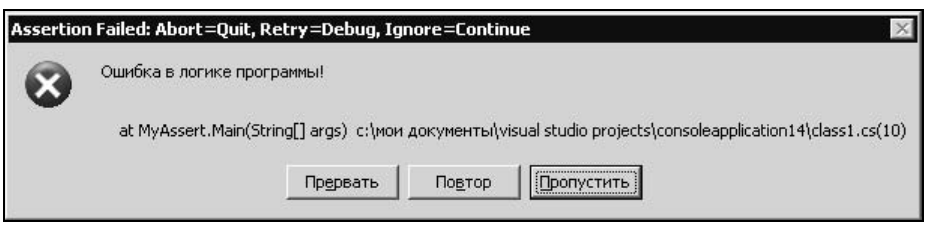


Рис. 19.1. Сообщение о том, что логика программы нарушена

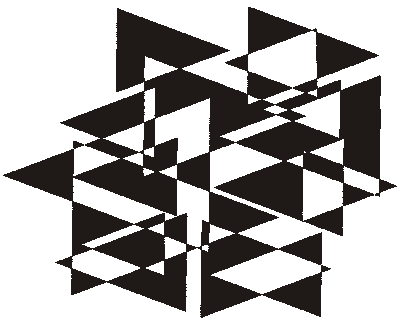
Откомпилируйте программу из листинга 19.5 с помощью следующей командной строки:

```
Csc /d:DEBUG MyAssert.cs
```

и вы увидите на экране окно, представленное на рис. 19.1

Итак, на этом я заканчиваю главу и всю книгу. Думаю, что читатель нашел в книге что-то полезное. Успехов в создании программ с помощью среды разработки Microsoft Visual Studio .NET 2003!

ЧАСТЬ VI



Приложения

В этой части содержится дополнительная информация.

Приложение 1. Список сайтов

Приложение 2. Редакции Visual Studio .NET 2003

Приложение 3. Системные требования для установки Visual Studio .NET 2003

Приложение 4. Языки .NET

Приложение 5. Общие типы данных языков .NET

Приложение 6. Сокращения и термины

ПРИЛОЖЕНИЕ 1

Список сайтов

Здесь приведены ссылки на сайты в сети Интернет, посвященные языку C# и технологии .NET.

Сайты, посвященные технологии .NET:

- **http://www.gotdotnet.com**
- **http://msdn.microsoft.com**
- **http://www.devx.com/dotnet/resources**

Сайты, посвященные языку C#:

- **http://www.icsharpcode.net**
- **http://www.c-sharpcorner.com**
- **http://www.csharphelp.com**
- **http://www.csharpindex.com**

ПРИЛОЖЕНИЕ 2

Редакции Visual Studio .NET 2003

Среда Visual Studio .NET 2003 поставляется конечному пользователю в четырех различных редакциях: Professional, Enterprise Developer, Enterprise Architect и Academic. В зависимости от редакции, среда Visual Studio .NET 2003 обладает теми или иными средствами и возможностями. В табл. П2.1 приведена сравнительная характеристика имеющихся редакций Visual Studio .NET 2003.

Таблица П2.1. *Различия редакций среды Visual Studio .NET 2003*

Средство/Возможность	Professional	Enterprise Developer	Enterprise Architect	Academic
Visual Basic .NET	Есть	Есть	Есть	Есть
Visual C++ .NET	Есть	Есть	Есть	Есть
Visual C# .NET	Есть	Есть	Есть	Есть
Создание и использование сервисов XML Web	Есть	Есть	Есть	Есть
Создание Web-приложений	Есть	Есть	Есть	Есть
Создание Windows-приложений	Есть	Есть	Есть	Есть
Указание дескриптора устройств	Есть	Есть	Есть	Есть
Создание таблиц и представлений с помощью SQL Server Desktop Engine	Есть	Есть	Есть	Есть

Таблица П2.1 *(окончание)*

Средство/Возможность	Professional	Enterprise Developer	Enterprise Architect	Academic
Создание таблиц, представлений, процедур, триггеров, функций и т. п. для SQL Server и Oracle	Нет	Есть	Есть	Нет
Visual SourceSafe	Нет	Есть	Есть	Нет
Тестирование сервисов XML Web и Web-приложений	Нет	Есть	Есть	Нет
Средства для студентов и преподавателей, включая Мастера создания приложений	Нет	Нет	Нет	Есть
Специальная документация и примеры кода для студентов и преподавателей	Нет	Нет	Нет	Есть

ПРИЛОЖЕНИЕ 3

Системные требования для установки Visual Studio .NET 2003

Для успешной работы со средой Visual Studio .NET 2003 корпорация Microsoft рекомендует соблюдать следующие аппаратные и программные требования (минимальные, в скобках указаны рекомендуемые требования для комфортной работы):

1. Центральный процессор компьютера — Pentium II 450 МГц (Pentium III 600 МГц).
2. Объем оперативной памяти — в зависимости от используемой операционной системы:
   - Windows 2000 Professional — 96 Мбайт (128 Мбайт);
   - Windows 2000 Server — 192 Мбайт (256 Мбайт);
   - Windows NT 4.0 Workstation — 64 Мбайт (96 Мбайт);
   - Windows NT 4.0 Server — 160 Мбайт (192 Мбайт);
   - Windows XP Professional — 160 Мбайт (192 Мбайт).
3. Место на жестком диске — 500 Мбайт на системном диске + 3 Гбайт на диске для установки среды (итого 3,5 Гбайт).
4. Операционная система — Windows 2000, Windows XP или Windows NT 4.0.

Примечание

Операционная система Windows NT 4.0 не поддерживает технологии ASP.NET, COM+, а также автоматическую "сборку мусора".

5. Привод CD-ROM или DVD-ROM.
6. Видеосистема с возможностью отображения 800×600 точек 256-ю цветами (1024×768 High Color 16-bit).
7. Мышь.

ПРИЛОЖЕНИЕ 4

Языки .NET

Здесь приводятся языки, для которых на сегодняшний день существуют компиляторы IL-кода. Напомню, что преобразование программы в IL-код позволяет выполнять такую программу в CLR.

В табл. П4.1 перечислены языки программирования .NET, которые поддерживаются корпорацией Microsoft. Информацию об этих языках вы можете получить по указанному в таблице адресу в сети Интернет.

Таблица П4.1. *Языки .NET, поддерживаемые корпорацией Microsoft*

Язык программирования	Адрес в Интернете
C++	**http://msdn.microsoft.com/vstudio/nextgen/technology/managedext.asp**
C#	**http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp**
Visual Basic .NET	**http://msdn.microsoft.com/vstudio/nextgen/technology/language.asp**
JScript	**http://msdn.microsoft.com/workshop/languages/clinic/scripting07142000.asp**

В табл. П4.2 перечислены языки программирования .NET, которые созданы не корпорацией Microsoft, но имеют свои собственные компиляторы для CLR. В таблице также указаны ссылки на адреса в Интернете, где вы можете получить дополнительную информацию о данных языках программирования. Замечу, что данный список далеко не полный. Число языков .NET постоянно увеличивается.

Таблица П4.2. *Другие языки .NET*

Язык программирования	Адрес в Интернете
APL	**http://www.dyadic.com**
COBOL	**http://www.adtools.com**
Component Pascal	**http://www2.fit.qut.edu.au**
Delta Forth	**http://www.dataman.ro**
Eiffel#	**http://www.eiffel.com**
Fortran	**http://www.lahey.com**
Haskell	**http://haskell.cs.yale.edu**
Mercury	**http://www.cs.mu.oz.au**
Mondrian	**http://www.mondrian-script.org**
Oberon	**http://www.oberon.ethz.ch**
Perl	**http://www.activestate.com**
Python	**http://www.activestate.com**
RPG	**http://www.asna.com**
Scheme	**http://rover.cs.nwu.edu**
SmallTalk	**http://www.qks.com**
Standart ML	**http://www.research.microsoft.com/Projects/SML.NET**
TMT Pascal	**http://www.tmt.com**

Если вы желаете узнать больше о языках .NET, обращайтесь за дополнительной информацией на сайт корпорации Microsoft.

ПРИЛОЖЕНИЕ 5

Общие типы данных языков .NET

Так как все языки среды Visual Studio .NET 2003 отличаются друг от друга (синтаксически), то каждый из них предоставляет программисту собственные ключевые слова для типов данных.

Например, если булево значение в Visual Basic .NET относится к типу `Boolean`, то в C# или C++ оно же будет относиться к типу `bool`. Такая же ситуация и с целыми числами. В Visual Basic есть тип `Integer`, а в C++ и C# данный тип обозначается ключевым словом `int`.

Однако в .NET Framework для всех простых типов данных предусмотрены общие ключевые слова. Например, для целого `Int32`, а для булевого значения `Boolean`.

В табл. П5.1 перечислены общие простые типы данных из пространства имен `System` среды .NET Framework.

Таблица П5.1. *Общие простые типы данных языков .NET Framework*

Тип данных	Краткое описание
`Boolean`	Булево значение. Может принимать одно из двух значений `true` или `false`
`Byte`	Целое беззнаковое восьмибитное число. Может принимать значения от 0 до 255
`Char`	Символьное значение. Может быть любым 16-битным символом Unicode
`DateTime`	Значение даты и времени
`Decimal`	Любое целое число в пределах от –79 228 162 514 264 337 593 543 950 335 до 79 228 162 514 264 337 593 543 950 335

Таблица П5.1 *(окончание)*

Тип данных	Краткое описание
`Double`	Число (64-битное) с плавающей точкой от –1.79769313486231570E308 до 1.79769313486231570E308
`Guid`	Служит для обозначения глобального уникального идентификатора (GUID)
`Int16`	16-битное целое число со знаком от –32 768 до 32 767
`Int32`	32-битное целое число со знаком от –2 147 483 648 до 2 147 483 647
`Int64`	64-битное целое число со знаком от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
`Single`	32-битное целое число с одинарной точностью от –3.40282346638528859E38 до 3.40282346638528859E38
`String`	Строка символов Unicode
`Void`	Используется только для описания функций, которые не возвращают никаких значений

Приведу еще несколько, не относящихся к простым типам контейнерных типов данных, общих для всех .NET-языков. Они перечислены в табл. П5.2.

Таблица П5.2. *Общие контейнерные типы данных языков .NET*

Тип данных	Краткое описание
`ArrayList`	Массив, динамически изменяющий свой размер
`BitArray`	Массив битовых значений, каждое из которых является булевым (`Boolean`)
`HashTable`	Коллекция связанных ключей и значений (упорядоченных по хэш-коду ключа)
`Queue`	Конструкция, по принципу работы обратная стеку. То есть "первым вошел — первым вышел"
`SortedList`	Аналог типа `HashTable`, но все элементы сортируются по фактическим ключам и доступны как по ключу, так и по индексу
`Stack`	Стековая конструкция. Принцип работы "первым вошел — последним вышел"

ПРИЛОЖЕНИЕ 6

Сокращения и термины

В этом приложении приведен список сокращений и терминов, с которыми вы можете столкнуться при чтении различных материалов и книг, посвященных технологии .NET. Основные сокращения и термины перечислены в табл. П6.1.

Таблица П6.1. *Основные сокращения и термины*

Сокращение/Термин	Краткое описание
.ASMX	Расширение файла с исходным текстом Web-службы
.ASPX	Расширение файла с исходным текстом ASP.NET
.CAB	Расширение сжатого файла
.config	Расширение конфигурационного файла .NET-приложения
ADO	ActiveX Data Objects, объекты данных ActiveX. Обеспечивают доступ к данным
ADO .NET	То же, что и ADO, но для доступа к данным в .NET
API	Application Programming Interface, программный интерфейс приложения
AppDomain	Область приложения
ASP	Active Server Pages, активные страницы сервера
ASP .NET	То же, что и ASP, но для технологии .NET
ATL	Active Template Library, библиотека активных шаблонов
BLOB	Binary Large Object, большой двоичный объект
CCW	COM Callable Wrapper, вызываемый упаковщик COM

Таблица П6.1 *(продолжение)*

Сокращение/Термин	Краткое описание
CLI	Common Language Infrastructure, общеязыковая инфраструктура
CLR	Common Language Runtime, общеязыковая среда выполнения
CLS	Common Language Specification, общеязыковая спецификация
CLSID	Идентификатор класса, используемый в технологии COM
COFF	Common Object File Format, общий объектный формат файлов
COM	Component Object Model, объектная модель компонентов
COM Interop	COM Interoperation, возможность взаимодействия с COM
COM+	Расширенная версия COM. В настоящее время используется новый термин .NET Framework
CTS	Common Type System, общая система типов
DB	Data Base, база данных
DCOM	Distributed Component Object Model, распределенная объектная модель компонентов
DHTML	Dynamic HyperText Markup Language, динамический язык разметки гипертекста
DISCO	Discovery of Web Service, открытие Web-службы
DISPID	Dispatch Identifier, диспетчерский идентификатор. Он используется в технологии COM для идентификации метода или свойства динамического вызова
DLL	Dynamically Linked Library, динамически присоединяемая библиотека
DNA	Distributed interNet Applications architecture, архитектура распределенных сетевых приложений
DOM	Document Object Model, объектная модель документа
DTD	DataType Document, документ типа данных
EXE	Расширение исполняемого файла
GC	Garbage Collector, "сборщик мусора"
GDI	Graphical Device Interface, интерфейс графического устройства

Таблица П6.1 *(продолжение)*

Сокращение/Термин	Краткое описание
GDI+	Библиотека, входящая в состав .NET, которая поддерживает расширенное управление графикой
Global.asax	Глобальный конфигурационный файл приложения ASP.NET
GUID	Globally Unique Identifier, глобальный универсальный идентификатор
HTML	HyperText Markup Language, язык разметки гипертекста
HTTP	HyperText Transfer Protocol, протокол передачи гипертекста
IDE	Integrated Development Environment, интегрированная среда разработки
IDL	Interface Definition Language, язык определения интерфейса
IE	Internet Explorer, проводник для сети Интернет
IID	Interface Identifier, идентификатор интерфейса
IIS	Internet Information Services, сервер
IL	Intermediate Language, промежуточный язык для технологии .NET
ILDASM	Intermediate Language Disassembler, дизассемблер промежуточного языка
Inproc	Внутрипроцессный
IP	Internet Protocol, протокол для сети Интернет
ISAPI	Information Server Application Programming Interface, прикладной интерфейс сервера IIS
Machine.config	Конфигурационный файл административной политики для компьютера
MBR	Marshal-By-Reference, маршаллинг по ссылке
MBV	Marshal-By-Value, маршаллинг по значению
MFC	Библиотека классов Microsoft Foundation Classes
MSI	Microsoft Windows Installer Package, пакет инсталляции для ОС Microsoft Windows
MSIL	Microsoft Intermediate Language, промежуточный язык Microsoft
MSVCRT	Microsoft Visual C++ Runtime, библиотека времени выполнения Microsoft Visual C++

Таблица П6.1 *(продолжение)*

Сокращение/Термин	Краткое описание
MSXML	Microsoft eXtensible Markup Language, расширяемый язык разметки Microsoft
MTS	Microsoft Transaction Server, сервер транзакций Microsoft
NTFS	NT FileSystem, файловая система NT
N-Tier	Многоуровневый
NTLM	NT Lan Manager
OBJREF	Object Reference, ссылка на объект
Out of proc	Внепроцессный
P/Invoke	Platform Invoke
PE	Portable Executable, импортируемый исполняемый файл
perm	Permissions, разрешения
RAD	Rapid Application Development, быстрая разработка приложений
RCW	Runtime Callable Wrapper, вызываемый упаковщик времени выполнения
REGASM	Register Assembly tool, утилита регистрации сборок
RPC	Remote Procedure Call, удаленный вызов процедур
SCL	SOAP Contract Language, язык контрактов SOAP
SDK	Software Development Kit, набор разработчика программного обеспечения
SEH	Structured Exception Handling, структурная обработка исключений
SMTP	Simple Mail Transfer Protocol, простой протокол передачи почты
SOAP	Simple Object Access Protocol, простой протокол доступа к объектам
SQL	Structured Query Language, язык структурированных запросов
Standard JIT	Оптимизированный двоичный код, включающий проверку IL
STL	Standard Template Library, библиотека стандартных шаблонов
TCP	Transport Control Protocol, транспортный управляющий протокол

Таблица П6.1 *(продолжение)*

Сокращение/Термин	Краткое описание
TLB	Type Library, библиотека типов
TLBEXP	Type Library Exporter Tool, утилита экспорта библиотек типов
TLBIMP	Type Library Importer Tool, утилита импорта библиотек типов
UDDI	Universal Description, Discovery and Integration Service. Независимая от платформы среда описания и открытия Web-служб
UDF	Uniform Data Format, унифицированный формат данных
UI	User Interface, интерфейс пользователя
URI	Uniform Resource Identifier, унифицированный идентификатор ресурса
URL	Uniform Resource Locator, унифицированный локатор ресурса
URT	Универсальная библиотека времени выполнения. В настоящее время не используется. Ее заменила .NET Framework
VB	Visual Basic
VBRUN	Visual Basic Runtime, библиотека времени выполнения Visual Basic
VES	Virtual Execution System, виртуальная система выполнения. Входит в состав CLR
VOS	Virtual Object System, виртуальная объектная система. Новое название CTS
VS.NET	Visual Studio .NET
WAP	Wireless Application Protocol, протокол беспроводных приложений
Web.config	Конфигурационный файл ASP.NET. В нем определяются HTTP-модули, управление состоянием сеанса и другие параметры ASP.NET
WML	Wireless Markup Language, язык гипертекстовой разметки для беспроводной связи
WSDL	Web Service Description Language, язык описания Web-служб
XML	Extensible Markup Language, расширяемый язык разметки

Таблица П6.1 *(окончание)*

Сокращение/Термин	Краткое описание
XPath	XML Path
XSD	XML Schema Definition, определение XML-схемы
XSL	Extensible Stylesheet Language, расширяемый язык таблиц стилей
XSLT	Extensible Stylesheet Language Transformation, язык преобразования расширяемых таблиц стилей

Приведенные в табл. П6.1 сокращения и термины составляют далеко не полный список. Дополнительную информацию по встретившимся незнакомым сокращениям вы всегда сможете найти в MSDN.

Предметный указатель

М

Н

О

П

P

C

T

У

Ф

Ц

Э

Я